EXHIBIT 1

UNITED STATES DISTRICT COURT
FOR THE WESTERN DISTRICT OF TEXAS
MIDLAND-ODESSA DIVISION

VIRTAMOVE, CORP., Plaintiff, v. GOOGLE LLC, Defendant.	Case No. 2:24-cv-00033-DC-DTG

PLAINTIFF VIRTAMOVE, CORP.'S CORRECTED PRELIMINARY DISCLOSURE OF ASSERTED CLAIMS AND INFRINGEMENT CONTENTIONS

I. OGP § I: Disclosure of Asserted Claims and Infringement Contentions

Pursuant to OGP § I, Plaintiff VirtaMove, Corp. submits the following Preliminary Disclosure of Asserted Claims and Infringement Contentions. This disclosure is based on the information available to VirtaMove as of the date of this disclosure, and VirtaMove reserves the right to amend this disclosure to the full extent permitted, consistent with the Court's Rules and Orders.

A. Asserted Claims

VirtaMove asserts that Defendant Google LLC ("Defendant" or "Google") infringes the following claims (collectively, "Asserted Claims"):

(1) U.S. Patent No. 7,519,814 ("the '814 patent"), claims 1, 2, 4, 6, 9, 10, 13, and 14; and

(2) U.S. Patent No. 7,784,058 ("the '058 patent"), claims 1–5, 10, and 18.

This Corrected Preliminary Disclosure of Asserted Claims and Infringement Contentions correctly reflects, consistent with the Complaint (Dkt. 1) and the Amended Complaint (Dkt. 27), that the only independent claims that are asserted in this case are independent claim 1 of the '814 patent

and independent claim 1 of the ’058 patent. Independent Claim 31 of the ’814 patent is not, was not, and will not be asserted in this case. This Corrected Preliminary Disclosure of Asserted Claims and Infringement Contentions also corrects the case caption and identification of the OGP as the applicable rule, and the cover pleading is corrected to reflect the identification of previously served and charted claims for the ’814 patent. Otherwise, the infringement theories remain identical to what was previously served on June 25, 2024.

B. Accused Instrumentalities of which VirtaMove is aware

VirtaMove asserts that the Asserted Claims are infringed by the various instrumentalities used, made, sold, offered for sale, or imported into the United States by Defendant, including certain (a) Google products and services using secure containerized applications, including without limitation Google Kubernetes Engine, Cloud Run, and Migrate to Containers, and all versions and variations thereof since the issuance of the ’814 patent; and (b) Google products and services using user mode critical system elements as shared libraries, including without limitation Google Kubernetes Engine, Cloud Run, and Migrate to Containers, and all versions and variations thereof since the issuance of the ’058 patent (“Accused Instrumentalities”). Defendant’s Accused Instrumentalities of which VirtaMove is presently aware are described in more detail in the accompanying preliminary infringement contention charts.

VirtaMove reserves the right to accuse additional products from Defendant to the extent VirtaMove becomes aware of additional products during the discovery process. Unless otherwise stated, VirtaMove’s assertions of infringement apply to all variations, versions, and applications of each of the Accused Instrumentalities, on information and belief, that different variations, versions, and applications of each of the Accused Instrumentalities are substantially the same for purposes of infringement of the Asserted Claims.

C. Claim Charts

VirtaMove's analysis of Defendant's products is based upon limited information that is publicly available, and based on VirtaMove's own investigation prior to any discovery in these actions. Specifically, VirtaMove's analysis is based on certain limited resources that evidence certain products made, sold, used, or imported into the United States by Defendant.

VirtaMove reserves the right to amend or supplement these disclosures for any of the following reasons:

(1) Defendant and/or third parties provide evidence relating to the Accused Instrumentalities;

(2) VirtaMove's position on infringement of specific claims may depend on the claim constructions adopted by the Court, which has not yet occured; and

(3) VirtaMove's investigation and analysis of Defendant's Accused Instrumentalities is based upon public information and VirtaMove's own investigations. VirtaMove reserves the right to amend these contentions based upon discovery of non-public information that VirtaMove anticipates receiving during discovery.

Attached and incorporated herein in their entirety, are charts identifying where each element of the Asserted Claims are found in the Accused Instrumentalities.

Unless otherwise indicated, the information provided that corresponds to each claim element is considered to indicate that each claim element is found within each of the different variations, versions, and applications of each of the respective Accused Instrumentalities described above.

D. Literal Infringement / Doctrine of Equivalents

With respect to the patents at issue, each element of each Asserted Claim is considered to be literally present. VirtaMove also contends that each Asserted Claim is infringed or has been infringed under the doctrine of equivalents in Defendant's Accused Instrumentalities. VirtaMove

also contends that Defendant both directly and indirectly infringes the Asserted Claims. For example, the Accused Instrumentalities are provided by the Defendant to customers, who are actively encouraged and instructed (for example, through Defendant's online instructions on its website and instructions, manual, or user guides that are provided with the Accused Instrumentalities) by Defendant to use the Accused Instrumentalities in ways that directly infringe the Asserted Claims. Defendant therefore specifically intends for and induces its customers to infringe the Asserted Claims under Section 271(b) through the customers' normal and customary use of the Accused Instrumentalities. In addition, Defendant is contributorily infringing the Asserted Claims under Section 271(c) and/or Section 271(f) by selling, offering for sale, or importing the Accused Instrumentalities into the United States, which constitute a material part of the inventions claimed in the Asserted Claims, are especially made or adapted to infringe the Asserted Claims, and are otherwise not staple articles or commodities of commerce suitable for non-infringing use.

E. Priority Dates

The Asserted Claims of the '814 patent are entitled to a priority date at least as early as September 15, 2003, the filing date of provisional application No. 60/502,619.

The Asserted Claims of the '058 patent are entitled to a priority date at least as early as September 22, 2003, the filing date of provisional application No. 60/504,213.

A diligent search continues for additional responsive information and VirtaMove reserves the right to supplement this response.

F. Identification of Instrumentalities Practicing the Claimed Invention

At this time, VirtaMove does not identify any of its instrumentalities as practicing the Asserted Claims. A diligent search continues for additional responsive information and VirtaMove reserves the right to supplement this response.

II. Document Production Accompanying Disclosure

Pursuant to Patent Rule 3-2, VirtaMove submits the following Document Production Accompanying Disclosure, along with an identification of the categories to which each of the documents corresponds.

A. Documents related to sale of the invention:

VirtaMove is presently unaware of any documents sufficient to evidence any discussion with, disclosure to, or other manner of providing to a third party, or sale of or offer to sell, the inventions recited in the Asserted Claims of the Asserted Patents prior to the application dates or priority dates for the Asserted Patents. A diligent search continues for such documents and VirtaMove reserves the right to supplement this response.

B. Documents related to conception:

VirtaMove identifies the following non-privileged documents as related to evidencing conception and reduction to practice of each claimed invention of the Asserted Patents: VM_GOOGLE_0000865–VM_GOOGLE_0000880. A diligent search continues for additional documents and VirtaMove reserves the right to supplement this response.

C. Documents including the file history:

VirtaMove identifies the following documents as being the file histories for the Asserted Patents: VM_GOOGLE_0000001–VM_GOOGLE_0000864.

Dated: July 1, 2024

Respectfully submitted,

/s/ Reza Mirzaie
Reza Mirzaie
CA State Bar No. 246953
Marc A. Fenster
CA State Bar No. 181067
Neil A. Rubin
CA State Bar No. 250761

Amy E. Hayden
CA State Bar No. 287026
Jacob R. Buczko
CA State Bar No. 269408
James S. Tsuei
CA State Bar No. 285530
James A. Milkey
CA State Bar No. 281283
Christian W. Conkle
CA State Bar No. 306374
Jonathan Ma
CA State Bar No. 312773
Daniel Kolko (CA SBN 341680)
RUSS AUGUST & KABAT
12424 Wilshire Boulevard, 12th Floor
Los Angeles, CA 90025
Telephone: 310-826-7474
Email: rmirzaie@raklaw.com
Email: mfenster@raklaw.com
Email: nrubin@raklaw.com
Email: ahayden@raklaw.com
Email: jbuczko@raklaw.com
Email: jtsuei@raklaw.com
Email: jmilkey@raklaw.com
Email: cconkle@raklaw.com
Email: jma@raklaw.com
Email: dkolko@raklaw.com

Qi (Peter) Tong
4925 Greenville Ave., Suite 200
Dallas, TX 75206
Email: ptong@raklaw.com

ATTORNEYS FOR PLAINTIFF VIRTAMOVE, CORP.

CERTIFICATE OF SERVICE

I certify that this document is being served upon counsel of record for Defendants on July 1, 2024 via e-mail.

/s/ Christian W. Conkle
Christian W. Conkle

U.S. Patent No. 7,519,814 ("'814 Patent")

Accused Instrumentalities: Google's products and services using user mode critical system elements as shared libraries, including without limitation Google Kubernetes Engine, Cloud Run, Migrate to Containers, and all versions and variations thereof since the issuance of the asserted patent.

Each Accused Instrumentality infringes the claims in substantially the same way, and the evidence shown in this chart is similarly applicable to each Accused Instrumentality. Each claim limitation is literally infringed by each Accused Instrumentality. However, to the extent any claim limitation is not met literally, it is nonetheless met under the doctrine of equivalents because the differences between the claim limitation and each Accused Instrumentality would be insubstantial, and each Accused Instrumentality performs substantially the same function, in substantially the same way, to achieve the same result as the claimed invention. Notably, Defendant has not yet articulated which, if any, particular claim limitations it believes are not met by the Accused Instrumentalities.

Claim 1

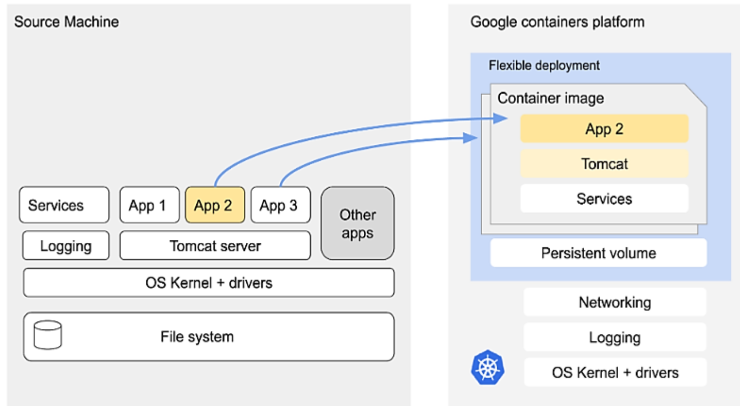
Claim 1	Accused Instrumentalities
[1pre] 1. In a system having a plurality of servers with operating systems that differ, operating in disparate computing environments, wherein each server includes a processor and an operating system including a kernel a set of associated local system files compatible with the processor, a method of providing at least some of the servers in the system with secure, executable, applications related to a service, wherein the applications are executed in a secure environment, wherein the applications each include	To the extent the preamble is limiting, Google and/or its customer practices, through the Accused Instrumentalities, in a system having a plurality of servers with operating systems that differ, operating in disparate computing environments, wherein each server includes a processor and an operating system including a kernel a set of associated local system files compatible with the processor, a method of providing at least some of the servers in the system with secure, executable, applications related to a service, wherein the applications are executed in a secure environment, wherein the applications each include an object executable by at least some of the different operating systems for performing a task related to the service, as claimed. For example, Google Kubernetes Engine and Cloud Run, as well as containers produced by Migrate to Containers, each runs on individual servers, each of which uses an independent operating system. Google provides and/or requires that each server includes a processor with one or more cores available to the OS kernel. Google further provides and/or requires that each server has a supported operating system (*e.g.*, Container-Optimized OS, Ubuntu), which includes a kernel and associated local system files, including for example libraries such as libc/glibc, configuration files, etc. On information and belief, there exist at least two GKE/Cloud Run servers that have different operating systems, for example Container-Optimized OS and Ubuntu. *See* claim limitations below.

Claim 1	Accused Instrumentalities
an object executable by at least some of the different operating systems for performing a task related to the service, the method comprising:	*See also, e.g.*: Google Kubernetes Engine (GKE) clusters provide secured and managed Kubernetes services with autoscaling and multi-cluster support. GKE lets you deploy, manage, and scale containerized applications on Kubernetes, powered by Google Cloud. https://cloud.google.com/migrate/containers/docs/getting-started This page describes the node images available for Google Kubernetes Engine (GKE) nodes. GKE Autopilot nodes always use Container-Optimized OS with containerd (`cos_containerd`), which is the recommended node operating system. If you use GKE Standard, you can choose the operating system image that runs on each node during cluster or node pool creation. You can also upgrade an existing Standard cluster to use a different node image. For instructions on how to set the node image, see Specifying a node image. https://cloud.google.com/kubernetes-engine/docs/concepts/node-images

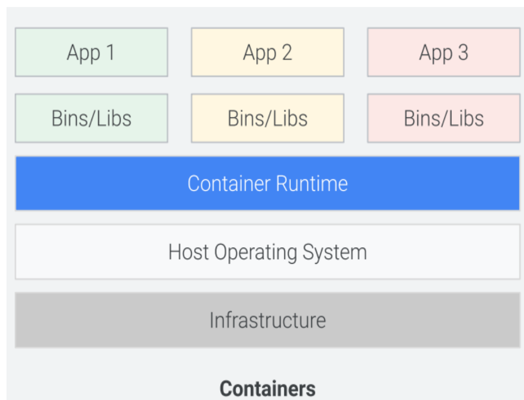
Claim 1	Accused Instrumentalities
	GKE offers the following node image options per OS for your cluster: **OS** / **Node images** **Container-Optimized OS** • Container-Optimized OS with containerd (`cos_containerd`) ★ GKE Autopilot clusters always use this image. • Container-Optimized OS with Docker (`cos`) (Unsupported in GKE version 1.24 and later) **Ubuntu** • Ubuntu with containerd (`ubuntu_containerd`) • Ubuntu with Docker (`ubuntu`) (Unsupported in GKE version 1.24 and later) **Windows Server** • Windows Server LTSC with containerd (`windows_ltsc_containerd`) (Supports both LTSC2022 and LTSC2019 node images) • Windows Server LTSC with Docker (`windows_ltsc`) (Unsupported in GKE version 1.24 and later. Unsupported for Windows Server LTSC2022.) **Warning:** Windows Server Semi-Annual Channel (SAC) images aren't supported after August 9, 2022 because Microsoft is removing support for the SAC. For potential impact and migration instructions, refer to Windows Server Semi-Annual Channel end of servicing. • Windows Server SAC with containerd (`windows_sac_containerd`) • Windows Server SAC with Docker (`windows_sac`) (Unsupported in GKE version 1.24 and later) https://cloud.google.com/kubernetes-engine/docs/concepts/node-images

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>
Use Migrate to Containers to modernize traditional applications away from virtual machine (VM) instances and into native containers that run on Google Kubernetes Engine (GKE), Anthos clusters, or Cloud Run platform. You can migrate workloads from VMs that run on VMware or Compute Engine, giving you the flexibility to containerize your existing workloads with ease.
https://cloud.google.com/migrate/containers/docs/getting-started
Given that, using tools like Migrate to Containers is a uniquely smart, efficient way to modernize traditional applications away from virtual machines and into native containers. Our unique automation approach extracts critical application elements from a VM so you can easily insert those elements into containers running on Google Kubernetes Engine (GKE), without artifacts like guest OS layers that VMs need but that are unnecessary for containers.
https://cloud.google.com/blog/products/containers-kubernetes/how-migrate-for-anthos-improves-vm-to-container-migration
Migrate to Containers supports migrations of VMs to containers on Google Kubernetes Engine on the 64-bit Linux operating systems listed in the following table.
<table>
<tr><th>OS</th><th>Compute Engine</th><th>VMware</th></tr>
<tr><td>CentOS</td><td>6.0, 7.0, 7.0 UEFI, 8.0</td><td>6.7, 6.9, 7.6</td></tr>
<tr><td>Debian</td><td>7.0, 8.0, 9.0, 10.0</td><td>9.4, 9.6</td></tr>
<tr><td>RHEL</td><td>6.0, 7.0, 7.0 UEFI, 7.4 SAP, 7.6 SAP, 8.0</td><td>6.5, 7.5, 7.6, 8.3</td></tr>
<tr><td>SUSE</td><td>12, 12 SP3 SAP, 12 SP4 SAP, 15, 15 SAP, 15 SP1 SAP</td><td>12 SP2, 12 SP3, 12 SP4, 15</td></tr>
<tr><td>Ubuntu</td><td>12 LTS, 14 LTS, 16 LTS, 16 LTS minimal, 18 LTS, 18 LTS minimal, 18 LTS UEFI, 19.04, 19.04 minimal</td><td>12.04.5 LTS, 14.04 LTS, 16.04 LTS, 18.04.10 LTS</td></tr>
</table>
https://cloud.google.com/migrate/containers/docs/compatible-os-versions, Last accessed on June 05, 2023
</td></tr>
</table>

Claim 1	Accused Instrumentalities
	Containers can run virtually anywhere, greatly easing development and deployment: on Linux, Windows, and Mac operating systems; on virtual machines or on physical servers; on a developer's machine or in data centers on-premises; and of course, in the public cloud. https://cloud.google.com/learn/what-are-containers A container is a way of packaging a given application's code and dependencies so that the application will run easily in any computing environment. This solves the common problem of https://services.google.com/fh/files/misc/why_container_security_matters.pdf

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>


https://cloud.google.com/blog/products/application-modernization/shift-your-apps-to-container-based-workloads-on-the-command-line



https://services.google.com/fh/files/misc/why_container_security_matters.pdf
</td></tr>
</table>

Claim 1	Accused Instrumentalities
	Containers virtualize CPU, memory, storage, and network resources at the operating system level, providing developers with a view of the OS logically isolated from other applications. https://cloud.google.com/learn/what-are-containers Containers are much more lightweight than VMs Containers virtualize at the OS level while VMs virtualize at the hardware level Containers share the OS kernel and use a fraction of the memory VMs require https://cloud.google.com/learn/what-are-containers Containers use specific features of the Linux kernel that "trick" individual applications into thinking they're in their own unique environment, even though multiple applications share the same host kernel. (If you're not familiar with the Linux kernel, it's a part of the operating system that communicates between processes--requests that do user tasks like opening a file, running a program-- and the hardware. It manages resources like memory and CPU to meet these requests). https://services.google.com/fh/files/misc/why_container_security_matters.pdf The core components of the Linux kernel that are used for containers are **cgroups** — control groups, which define the resources like CPU and memory which are available to a given process — and **namespaces**, which are a way of separating processes by restricting what each process can see, so that system resources "appear" isolated to the process. https://services.google.com/fh/files/misc/why_container_security_matters.pdf
[1a] storing in memory accessible to at least some of the servers a plurality of secure containers of application	The method practiced by Google and/or its customer through the Accused Instrumentalities includes a step of storing in memory accessible to at least some of the servers a plurality of secure containers of application software, each container comprising one or more of the executable applications and a

Claim 1	Accused Instrumentalities
software, each container comprising one or more of the executable applications and a set of associated system files required to execute the one or more applications, for use with a local kernel residing permanently on one of the servers;	set of associated system files required to execute the one or more applications, for use with a local kernel residing permanently on one of the servers. For example, GKE, Cloud Run, and Migrate to Containers each stores application containers, sometimes called Docker containers, container images, Kubernetes containers, or Kubernetes pods, in persistent storage available to each node running the application. The container might be in a format defined by the Open Container Initiative. This storage may be physically attached to the server or connected through any supported interconnect, including over a network. Each container includes the application software as well as a Linux user space required to execute the application, for example libc/glibc and other shared libraries, configuration files, etc. necessary for the application. For example, the container includes a base OS image, provided by Google or by a third party, such as a Debian, Rocky Linux, or Ubuntu base image. The container is compatible with the host kernel, for example because the container libraries are linked against the Linux kernel, and the supported host operating systems also use the Linux kernel, which has a stable binary interface. For another example, GKE and Cloud Run each stores files, pertaining to the applications, in ephemeral or persistent volumes, required to execute the applications within those containers. Because these volumes are stored and accessible within the GKE/Cloud Run environment, it is inferred that they are stored in the memory of the server as claimed. *See, e.g.*:

Claim 1	Accused Instrumentalities
	## What are base images? A base image is the starting point for most container-based development workflows. Developers start with a base image and layer on top of it the necessary libraries, binaries, and configuration files used to run their application. Many base images are basic or minimal Linux distributions: Debian, Ubuntu, Red Hat Enterprise Linux (RHEL), Rocky Linux, or Alpine. Developers can consume these images directly from Docker Hub or other sources. There are official providers along with a wide variety of other downstream repackagers that layer software to meet customer needs. Google maintains base images for building its own applications. These images are built from the same source that Docker Hub uses. Therefore, they match the images you would get from Docker Hub. The advantage of using Google-maintained images is that they are stored on Google Cloud, so you can pull these images directly from your environment without having to traverse networks. Google updates these images whenever a new version of an official image is released. For more information on image versions, see the GitHub repository of official images. https://cloud.google.com/software-supply-chain-security/docs/base-images

Claim 1	Accused Instrumentalities

Google-provided base images

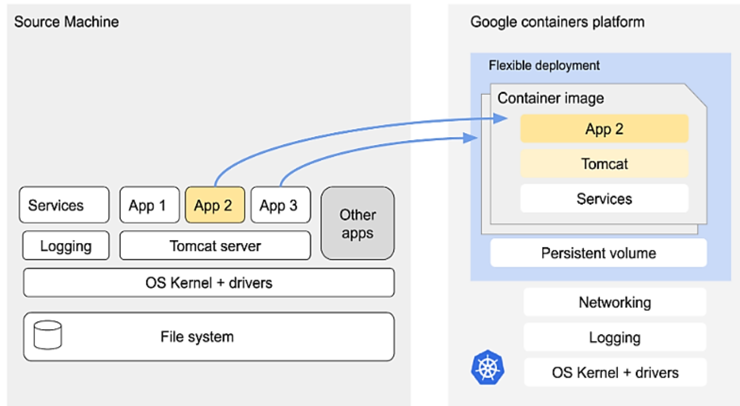
Google-provided base images are available for the following OS distributions:

OS	Repository path	Google Cloud Marketplace listing
Debian 10 "Buster"	`marketplace.gcr.io/google/debian10`	Google Cloud Marketplace
Debian 11 "Bullseye"	`marketplace.gcr.io/google/debian11`	Google Cloud Marketplace
Debian 12 "Bookworm"	`marketplace.gcr.io/google/debian12`	Google Cloud Marketplace
Rocky Linux 8	`marketplace.gcr.io/google/rockylinux8`	Google Cloud Marketplace
Rocky Linux 9	`marketplace.gcr.io/google/rockylinux9`	Google Cloud Marketplace
Ubuntu 20.04	`marketplace.gcr.io/google/ubuntu2004`	Google Cloud Marketplace
Ubuntu 22.04	`marketplace.gcr.io/google/ubuntu2204`	Google Cloud Marketplace
Ubuntu 24.04	`marketplace.gcr.io/google/ubuntu2404`	Google Cloud Marketplace

https://cloud.google.com/software-supply-chain-security/docs/base-images

Claim 1	Accused Instrumentalities
	There are several storage options for applications running on Google Kubernetes Engine (GKE). The choices vary in terms of Volumes are a storage unit accessible to containers in a Pod. Some volume types are backed by ephemeral storage. Ephemeral storage types (for example, emptyDir) do not persist after the Pod ceases to exist. These types are useful for scratch space for applications. You can manage your local ephemeral storage resources as you do your CPU and memory resources. Other volume types are backed by durable storage. https://cloud.google.com/kubernetes-engine/docs/concepts/storage-overview 6. **Do Docker containers package up the entire OS and make it easier to deploy?** Docker containers do not package up the OS. They package up the applications with everything that the application needs to run. The engine is installed on top of the OS running on a host. Containers share the OS kernel allowing a single host to run multiple containers. https://www.docker.com/blog/the-10-most-common-questions-it-admins-ask-about-docker/ At its core, a volume is a directory, possibly with some data in it, which is accessible to the containers in a pod. How that directory comes to be, the `.spec.containers[*].volumeMounts`. A process in a container sees a filesystem view composed from the initial contents of the container image, plus volumes (if defined) mounted inside the container. The process sees a root filesystem that initially matches the contents of the container image. Any writes to within that filesystem hierarchy, if allowed, affect what that process views when it performs a subsequent filesystem access. Volumes mount at the specified paths within the image. For each container defined within a Pod, you must independently specify where to mount each volume that the container uses. https://kubernetes.io/docs/concepts/storage/volumes/

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>



https://cloud.google.com/blog/products/application-modernization/shift-your-apps-to-container-based-workloads-on-the-command-line

A container is a way of packaging a given application's code and dependencies so that the application will run easily in any computing environment. This solves the common problem of

The container image specifies the container's file system. For example, if you're running a Node.js application, the container image would contain your app, Node.js, and other dependencies like Linux system libraries (except the kernel). A container image usually extends a base operating system image, or **base image**. This base image is the basis of your container, so you'll want to ensure that it's properly patched and free from known vulnerabilities.

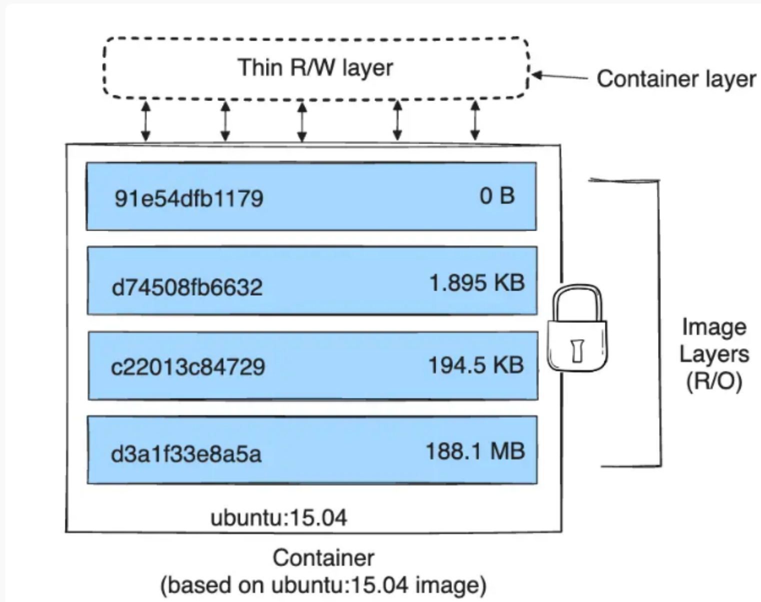
</td></tr>
</table>

Claim 1	Accused Instrumentalities
	workloads onto each server. As such, the architecture of containers means that they're deployed with multiple containers sharing the same kernel. https://services.google.com/fh/files/misc/why_container_security_matters.pdf Containers are lightweight packages of your application code together with dependencies such as specific versions of programming language runtimes and libraries required to run your software services. https://cloud.google.com/learn/what-are-containers

Claim 1	Accused Instrumentalities
	About storage drivers To use storage drivers effectively, it's important to know how Docker builds and stores images, and how these images are used by containers. You can use this information to make informed choices about the best way to persist data from your applications and avoid performance problems along the way. **Storage drivers versus Docker volumes** Docker uses storage drivers to store image layers, and to store data in the writable layer of a container. The container's writable layer doesn't persist after the container is deleted, but is suitable for storing ephemeral data that is generated at runtime. Storage drivers are optimized for space efficiency, but (depending on the storage driver) write speeds are lower than native file system performance, especially for storage drivers that use a copy-on-write filesystem. Write-intensive applications, such as database storage, are impacted by a performance overhead, particularly if pre-existing data exists in the read-only layer. Use Docker volumes for write-intensive data, data that must persist beyond the container's lifespan, and data that must be shared between containers. Refer to the volumes section to learn how to use volumes to persist data and improve performance. https://docs.docker.com/storage/storagedriver/

Claim 1	Accused Instrumentalities

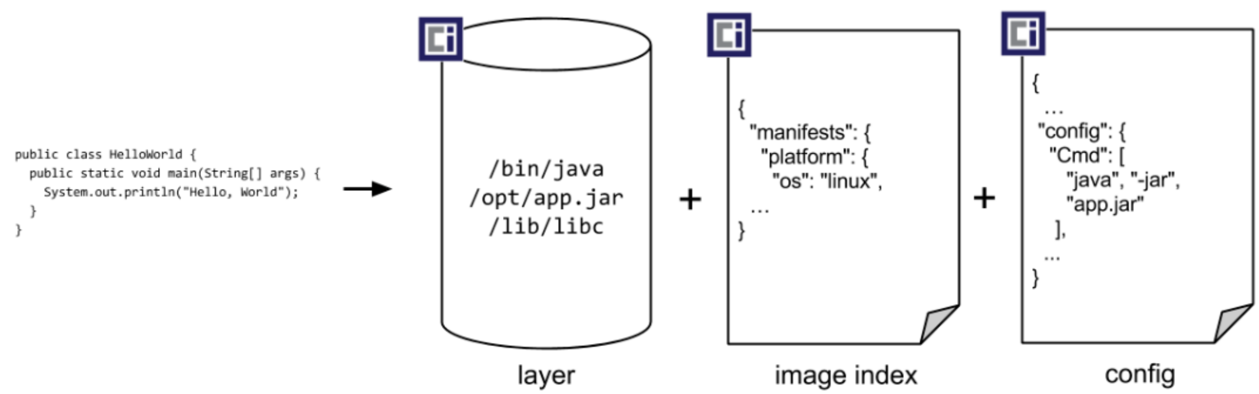
	# Images and layers
	
	A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile:
	
	`# syntax=docker/dockerfile:1`
	
	`FROM ubuntu:22.04`
	`LABEL org.opencontainers.image.authors="org@example.com"`
	`COPY . /app`
	`RUN make /app`
	`RUN rm -r $HOME/.cache`
	`CMD python /app/app.py`
	
	This Dockerfile contains four commands. Commands that modify the filesystem create a layer. The `FROM` statement starts out by creating a layer from the `ubuntu:22.04` image. The `LABEL` command only modifies the image's metadata, and doesn't produce a new layer. The `COPY` command adds some files from your Docker client's current directory. The first `RUN` command builds your application using the `make` command, and writes the result to a new layer. The second `RUN` command removes a cache directory, and writes the result to a new layer. Finally, the `CMD` instruction specifies what command to run within the container, which only modifies the image's metadata, which doesn't produce an image layer.
	
	https://docs.docker.com/storage/storagedriver/

Claim 1	Accused Instrumentalities
	Each layer is only a set of differences from the layer before it. Note that both *adding*, and *removing* files will result in a new layer. In the example above, the `$HOME/.cache` directory is removed, but will still be available in the previous layer and add up to the image's total size. Refer to the Best practices for writing Dockerfiles and use multi-stage builds sections to learn how to optimize your Dockerfiles for efficient images. The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The diagram below shows a container based on an `ubuntu:15.04` image.  https://docs.docker.com/storage/storagedriver/

Claim 1	Accused Instrumentalities
	# Volumes Volumes are the preferred mechanism for persisting data generated by and used by Docker containers. While bind mounts are dependent on the directory structure and OS of the host machine, volumes are completely managed by Docker. Volumes have several advantages over bind mounts: https://kubernetes.io/docs/concepts/storage/volumes/ ## Container environment The Kubernetes Container environment provides several important resources to Containers: • A filesystem, which is a combination of an image and one or more volumes. • Information about the Container itself. • Information about other objects in the cluster. https://kubernetes.io/docs/concepts/containers/container-environment/

Claim 1	Accused Instrumentalities
	Images A container image represents binary data that encapsulates an application and all its software dependencies. Container images are executable software bundles that can run standalone and that make very well defined assumptions about their runtime environment. You typically create a container image of your application and push it to a registry before referring to it in a Pod. https://kubernetes.io/docs/concepts/containers/images/ **Volumes** On-disk files in a container are ephemeral, which presents some problems for non-trivial applications when running in containers. One problem occurs when a container crashes or is stopped. Container state is not saved so all of the files that were created or modified during the lifetime of the container are lost. During a crash, kubelet restarts the container with a clean state. Another problem occurs when multiple containers are running in a `Pod` and need to share files. It can be challenging to setup and access a shared filesystem across all of the containers. The Kubernetes volume abstraction solves both of these problems. Familiarity with Pods is suggested. https://kubernetes.io/docs/concepts/storage/volumes/

Claim 1	Accused Instrumentalities
	Open Container Initiative **Image Format Specification** This specification defines an OCI Image, consisting of an image manifest, an image index (optional), a set of filesystem layers, and a configuration. The goal of this specification is to enable the creation of interoperable tools for building, transporting, and preparing a container image to run. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

Claim 1	Accused Instrumentalities
	## Overview At a high level the image manifest contains metadata about the contents and dependencies of the image including the content-addressable identity of one or more filesystem layer changeset archives that will be unpacked to make up the final runnable filesystem. The image configuration includes information such as application arguments, environments, etc. The image index is a higher-level manifest which points to a list of manifests and descriptors. Typically, these manifests may provide different implementations of the image, possibly varying by platform or other attributes.  https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

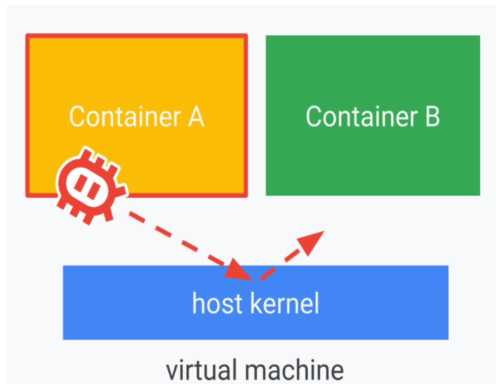
Claim 1	Accused Instrumentalities
	# OCI Image Configuration An OCI *Image* is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime. This specification outlines the JSON format describing images for use with a container runtime and execution tool and its relationship to filesystem changesets, described in Layers. This section defines the `application/vnd.oci.image.config.v1+json` media type. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

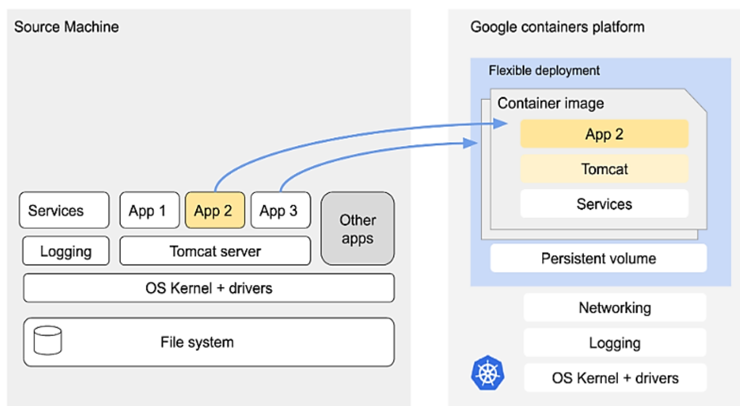
Claim 1	Accused Instrumentalities
	Layer • Image filesystems are composed of *layers*. • Each layer represents a set of filesystem changes in a tar-based layer format, recording files to be added, changed, or deleted relative to its parent layer. • Layers do not have configuration metadata such as environment variables or default arguments - these are properties of the image as a whole rather than any particular layer. • Using a layer-based or union filesystem such as AUFS, or by computing the diff from filesystem snapshots, the filesystem changeset can be used to present a series of image layers as if they were one cohesive filesystem. **Image JSON** • Each image has an associated JSON structure which describes some basic information about the image such as date created, author, as well as execution/runtime configuration like its entrypoint, default arguments, networking, and volumes. • The JSON structure also references a cryptographic hash of each layer used by the image, and provides history information for those layers. • This JSON is considered to be immutable, because changing it would change the computed ImageID. • Changing it means creating a new derived image, instead of changing the existing image. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

Claim 1	Accused Instrumentalities
	• **rootfs** *object*, REQUIRED The rootfs key references the layer content addresses used by the image. This makes the image config hash depend on the filesystem hash. ◦ **type** *string*, REQUIRED MUST be set to `layers`. Implementations MUST generate an error if they encounter a unknown value while verifying or unpacking an image. ◦ **diff_ids** *array of strings*, REQUIRED An array of layer content hashes (`DiffIDs`), in order from first to last. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md
[1b] wherein the set of associated system files are compatible with a local kernel of at least some of the plurality of different operating systems,	In the method practiced by Google through the Accused Instrumentalities, the set of associated system files are compatible with a local kernel of at least some of the plurality of different operating systems. The system files in the container are compatible with the host kernel, for example because they are linked against the Linux kernel and the supported host operating systems also use the Linux kernel, which has a stable binary interface. *See* discussion and evidence in element [1a] above. *See also, e.g.*:

Claim 1	Accused Instrumentalities
	A container is a way of packaging a given application's code and dependencies so that the application will run easily in any computing environment. This solves the common problem of The container image specifies the container's file system. For example, if you're running a Node.js application, the container image would contain your app, Node.js, and other dependencies like Linux system libraries (except the kernel). A container image usually extends a base operating system image, or **base image**. This base image is the basis of your container, so you'll want to ensure that it's properly patched and free from known vulnerabilities. Containers use specific features of the Linux kernel that "trick" individual applications into thinking they're in their own unique environment, even though multiple applications share the same host kernel. (If you're not familiar with the Linux kernel, it's a part of the operating system that communicates between processes--requests that do user tasks like opening a file, running a program-- and the hardware. It manages resources like memory and CPU to meet these requests). https://services.google.com/fh/files/misc/why_container_security_matters.pdf

Claim 1	Accused Instrumentalities
	Containers can run virtually anywhere, greatly easing development and deployment: on Linux, Windows, and Mac operating systems; on virtual machines or on physical servers; on a developer's machine or in data centers on-premises; and of course, in the public cloud. https://cloud.google.com/learn/what-are-containers
[1c] the containers of application software excluding a kernel,	In the method practiced by Google and/or its customer through the Accused Instrumentalities, the containers of application software exclude a kernel. *See* discussion and evidence in element [1a] above. *See also, e.g.*: • **Higher utilization and density**, leveraging automatic bin-packing and auto-scaling capabilities, Kubernetes places containers optimally in nodes based on required resources while scaling as needed, without impairing availability. In addition, unlike VMs, all containers on a single node share one copy of the operating system and don't each require their own OS image and vCPU, resulting in a much smaller memory footprint and CPU needs. This means more workloads running on fewer compute resources. https://cloud.google.com/blog/products/containers-kubernetes/how-migrate-for-anthos-improves-vm-to-container-migration

Claim 1	Accused Instrumentalities
	workloads onto each server. As such, the architecture of containers means that they're deployed with multiple containers sharing the same kernel.  https://services.google.com/fh/files/misc/why_container_security_matters.pdf 6. **Do Docker containers package up the entire OS and make it easier to deploy?** Docker containers do not package up the OS. They package up the applications with everything that the application needs to run. The engine is installed on top of the OS running on a host. Containers share the OS kernel allowing a single host to run multiple containers. https://www.docker.com/blog/the-10-most-common-questions-it-admins-ask-about-docker/
[1d] wherein some or all of the associated system files within a container stored in memory are utilized in place of the associated local system files that remain resident on the server,	In the method practiced by Google and/or its customer through the Accused Instrumentalities, some or all of the associated system files within a container stored in memory are utilized in place of the associated local system files that remain resident on the server. For example, each container will utilize its own local system files, including libraries such as libc/glibc and configuration files, not the corresponding libraries and configuration files of the host OS. *See* discussion and evidence in element [1a] above. *See also, e.g.*:

Claim 1	Accused Instrumentalities
	One of the primary reasons to adopt containers is for your applications to be decoupled from the underlying environment and support higher resource utilization by "bin packing" multiple workloads onto each server. As such, the architecture of containers means that they're deployed with multiple containers sharing the same kernel. The container image specifies the container's file system. For example, if you're running a Node.js application, the container image would contain your app, Node.js, and other dependencies like Linux system libraries (except the kernel). A container image usually extends a base operating system image, or **base image**. This base image is the basis of your container, so you'll want to ensure that it's properly patched and free from known vulnerabilities. https://services.google.com/fh/files/misc/why_container_security_matters.pdf  https://cloud.google.com/blog/products/application-modernization/shift-your-apps-to-container-based-workloads-on-the-command-line

Claim 1	Accused Instrumentalities
[1e] wherein said associated system files utilized in place of the associated local system files are copies or modified copies of the associated local system files that remain resident on the server,	In the method practiced by Google and/or its customer through the Accused Instrumentalities, said associated system files utilized in place of the associated local system files are copies or modified copies of the associated local system files that remain resident on the server. For example, in some cases the host OS and container will use one or more identical system files, for example when both the host and the container incorporate the same Linux distribution version, or when both host and container use the same version of libc. In other cases modified copies are used instead, for example when different versions of the same library, or configuration files with different parameters, are used by the host and container. *See* discussion and evidence in element [1a] above. *See also, e.g.*: One of the primary reasons to adopt containers is for your applications to be decoupled from the underlying environment and support higher resource utilization by "bin packing" multiple workloads onto each server. As such, the architecture of containers means that they're deployed with multiple containers sharing the same kernel. The container image specifies the container's file system. For example, if you're running a Node.js application, the container image would contain your app, Node.js, and other dependencies like Linux system libraries (except the kernel). A container image usually extends a base operating system image, or **base image**. This base image is the basis of your container, so you'll want to ensure that it's properly patched and free from known vulnerabilities. https://services.google.com/fh/files/misc/why_container_security_matters.pdf

Claim 1	Accused Instrumentalities
	`COPY` and `ADD`: These commands copy files and directories from your local filesystem into the Docker image. They are often used to include your application code, configuration files, and dependencies. https://medium.com/@swalperen3008/what-is-dockerize-and-dockerize-your-project-a-step-by-step-guide-899c48a34df6 Container images A container image is a ready-to-run software package containing everything needed to run an application: the code and any runtime it requires, application and system libraries, and default values for any essential settings. https://kubernetes.io/docs/concepts/containers/

Claim 1	Accused Instrumentalities
	# About storage drivers To use storage drivers effectively, it's important to know how Docker builds and stores images, and how these images are used by containers. You can use this information to make informed choices about the best way to persist data from your applications and avoid performance problems along the way. ## Storage drivers versus Docker volumes Docker uses storage drivers to store image layers, and to store data in the writable layer of a container. The container's writable layer doesn't persist after the container is deleted, but is suitable for storing ephemeral data that is generated at runtime. Storage drivers are optimized for space efficiency, but (depending on the storage driver) write speeds are lower than native file system performance, especially for storage drivers that use a copy-on-write filesystem. Write-intensive applications, such as database storage, are impacted by a performance overhead, particularly if pre-existing data exists in the read-only layer. Use Docker volumes for write-intensive data, data that must persist beyond the container's lifespan, and data that must be shared between containers. Refer to the volumes section to learn how to use volumes to persist data and improve performance. https://docs.docker.com/storage/storagedriver/

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>

Images and layers

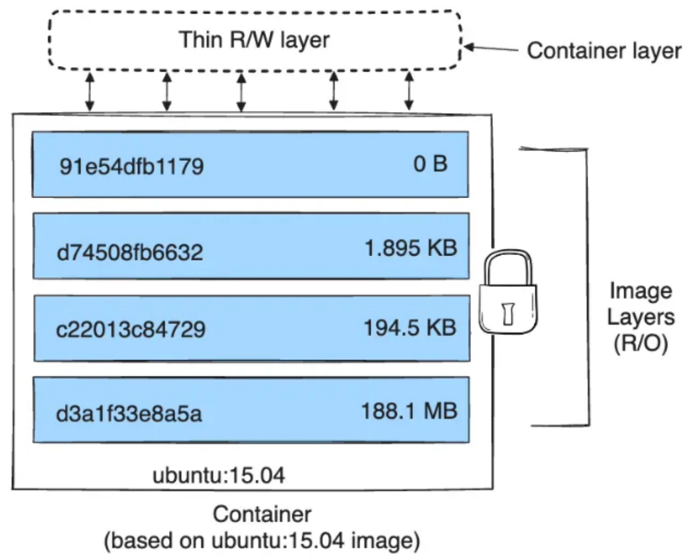
A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile:

```
# syntax=docker/dockerfile:1

FROM ubuntu:22.04
LABEL org.opencontainers.image.authors="org@example.com"
COPY . /app
RUN make /app
RUN rm -r $HOME/.cache
CMD python /app/app.py
```

This Dockerfile contains four commands. Commands that modify the filesystem create a layer. The `FROM` statement starts out by creating a layer from the `ubuntu:22.04` image. The `LABEL` command only modifies the image's metadata, and doesn't produce a new layer. The `COPY` command adds some files from your Docker client's current directory. The first `RUN` command builds your application using the `make` command, and writes the result to a new layer. The second `RUN` command removes a cache directory, and writes the result to a new layer. Finally, the `CMD` instruction specifies what command to run within the container, which only modifies the image's metadata, which doesn't produce an image layer.

https://docs.docker.com/storage/storagedriver/
</td></tr>
</table>

Claim 1	Accused Instrumentalities
	Each layer is only a set of differences from the layer before it. Note that both *adding*, and *removing* files will result in a new layer. In the example above, the `$HOME/.cache` directory is removed, but will still be available in the previous layer and add up to the image's total size. Refer to the Best practices for writing Dockerfiles and use multi-stage builds sections to learn how to optimize your Dockerfiles for efficient images. The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The diagram below shows a container based on an `ubuntu:15.04` image.  https://docs.docker.com/storage/storagedriver/

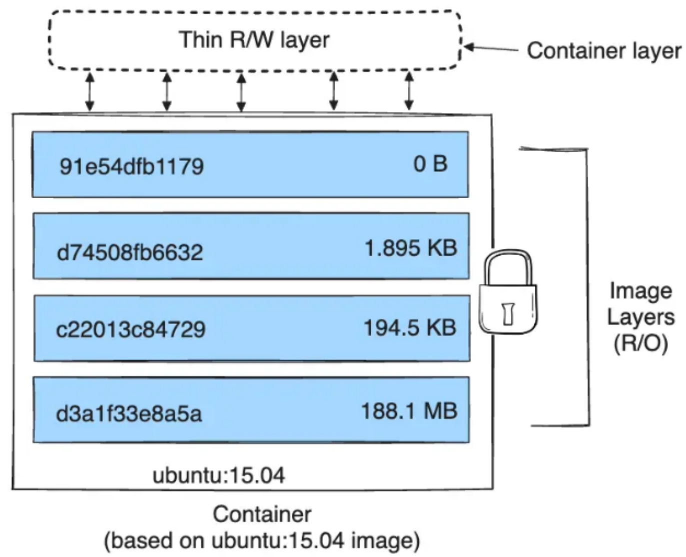
Claim 1	Accused Instrumentalities
[1f] and wherein the application software cannot be shared between the plurality of secure containers of application software,	In the method practiced by Google through the Accused Instrumentalities, the application software cannot be shared between the plurality of secure containers of application software. For example, each container has an isolated runtime environment that cannot be accessed by other containers, for example including a per-container writeable layer or other ephemeral per-container storage. For another example, when the plurality of secure containers each corresponds to a different container image, each container cannot access another container's image and therefore application software. *See, e.g.*: Containers use specific features of the Linux kernel that "trick" individual applications into thinking they're in their own unique environment, even though multiple applications share the same host kernel. (If you're not familiar with the Linux kernel, it's a part of the operating system that communicates between processes--requests that do user tasks like opening a file, running a program-- and the hardware. It manages resources like memory and CPU to meet these requests). https://services.google.com/fh/files/misc/why_container_security_matters.pdf The core components of the Linux kernel that are used for containers are **cgroups** — control groups, which define the resources like CPU and memory which are available to a given process — and **namespaces**, which are a way of separating processes by restricting what each process can see, so that system resources "appear" isolated to the process. https://services.google.com/fh/files/misc/why_container_security_matters.pdf reason. Furthermore, files within a container are inaccessible to other containers running in the same Pod. The Kubernetes https://cloud.google.com/kubernetes-engine/docs/concepts/volumes

Claim 1	Accused Instrumentalities
	A *Pod* (as in a pod of whales or pea pod) is a group of one or more containers, with shared storage and network resources, and a specification for how to run the containers. A Pod's contents are always co-located and co-scheduled, and run in a shared context. A Pod models an application-specific "logical host": it contains one or more application containers which are relatively tightly coupled. In non-cloud contexts, applications executed on the same physical or virtual machine are analogous to cloud applications executed on the same logical host. The shared context of a Pod is a set of Linux namespaces, cgroups, and potentially other facets of isolation - the same things that isolate a container. Within a Pod's context, the individual applications may have further sub-isolations applied. https://kubernetes.io/docs/concepts/workloads/pods/ ranges can access. GKE Sandbox for the Standard mode of operation provides a second layer of defense between containerized workloads on GKE for enhanced workload security. GKE https://cloud.google.com/kubernetes-engine#section-2

Claim 1	Accused Instrumentalities
	About storage drivers To use storage drivers effectively, it's important to know how Docker builds and stores images, and how these images are used by containers. You can use this information to make informed choices about the best way to persist data from your applications and avoid performance problems along the way. **Storage drivers versus Docker volumes** Docker uses storage drivers to store image layers, and to store data in the writable layer of a container. The container's writable layer doesn't persist after the container is deleted, but is suitable for storing ephemeral data that is generated at runtime. Storage drivers are optimized for space efficiency, but (depending on the storage driver) write speeds are lower than native file system performance, especially for storage drivers that use a copy-on-write filesystem. Write-intensive applications, such as database storage, are impacted by a performance overhead, particularly if pre-existing data exists in the read-only layer. Use Docker volumes for write-intensive data, data that must persist beyond the container's lifespan, and data that must be shared between containers. Refer to the volumes section to learn how to use volumes to persist data and improve performance. https://docs.docker.com/storage/storagedriver/

Claim 1	Accused Instrumentalities

	Images and layers
	
	A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile:
	
	`# syntax=docker/dockerfile:1`
	
	`FROM ubuntu:22.04`
	`LABEL org.opencontainers.image.authors="org@example.com"`
	`COPY . /app`
	`RUN make /app`
	`RUN rm -r $HOME/.cache`
	`CMD python /app/app.py`
	
	This Dockerfile contains four commands. Commands that modify the filesystem create a layer. The `FROM` statement starts out by creating a layer from the `ubuntu:22.04` image. The `LABEL` command only modifies the image's metadata, and doesn't produce a new layer. The `COPY` command adds some files from your Docker client's current directory. The first `RUN` command builds your application using the `make` command, and writes the result to a new layer. The second `RUN` command removes a cache directory, and writes the result to a new layer. Finally, the `CMD` instruction specifies what command to run within the container, which only modifies the image's metadata, which doesn't produce an image layer.
	
	https://docs.docker.com/storage/storagedriver/

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>
Each layer is only a set of differences from the layer before it. Note that both adding, and removing files will result in a new layer. In the example above, the <code>$HOME/.cache</code> directory is removed, but will still be available in the previous layer and add up to the image's total size. Refer to the Best practices for writing Dockerfiles and use multi-stage builds sections to learn how to optimize your Dockerfiles for efficient images.
The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The diagram below shows a container based on an <code>ubuntu:15.04</code> image.


https://docs.docker.com/storage/storagedriver/
</td></tr>
</table>

Claim 1	Accused Instrumentalities
[1g] and wherein each of the containers has a unique root file system that is different from an operating system's root file system.	In the method practiced by Google through the Accused Instrumentalities, each of the containers has a unique root file system that is different from an operating system's root file system. For example, the container's root file system comprises the image layer(s), an ephemeral writeable layer (e.g., in Docker terminology the container layer), and optionally one or more volumes. This root file system is distinct and isolated from the host operating system's root file system. *See, e.g.*: The original purpose of the cgroup, `chroot`, and namespace facilities in the kernel was to protect applications from noisy, nosey, and messy neighbors. Combining these with container images created an abstraction that also isolates applications from the (heterogeneous) operating systems on which they run. This decoupling of image and OS makes it possible to provide the same deployment environment in both development and production, which, in turn, improves deployment reliability and speeds up development by reducing inconsistencies and friction. "Borg, Omega, and, Kubernetes," https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/44843.pdf In Docker and Kubernetes, the container's root filesystem (rootfs) is based on the filesystem packaged with the image. The image's filesystem is immutable. Any change a container makes to the rootfs is stored separately and is destroyed with the container. This way, the image's filesystem https://opensource.googleblog.com/2023/04/gvisor-improves-performance-with-root-filesystem-overlay.html

Claim 1	Accused Instrumentalities
	To use a volume, specify the volumes to provide for the Pod in `.spec.volumes` and declare where to mount those volumes into containers in `.spec.containers[*].volumeMounts`. A process in a container sees a filesystem view composed from the initial contents of the container image, plus volumes (if defined) mounted inside the container. The process sees a root filesystem that initially matches the contents of the container image. Any writes to within that filesystem hierarchy, if allowed, affect what that process views when it performs a subsequent filesystem access. Volumes mount at the specified paths within the image. For each container defined within a Pod, you must independently specify where to mount each volume that the container uses. https://kubernetes.io/docs/concepts/storage/volumes/

Claim 1	Accused Instrumentalities
	About storage drivers To use storage drivers effectively, it's important to know how Docker builds and stores images, and how these images are used by containers. You can use this information to make informed choices about the best way to persist data from your applications and avoid performance problems along the way. **Storage drivers versus Docker volumes** Docker uses storage drivers to store image layers, and to store data in the writable layer of a container. The container's writable layer doesn't persist after the container is deleted, but is suitable for storing ephemeral data that is generated at runtime. Storage drivers are optimized for space efficiency, but (depending on the storage driver) write speeds are lower than native file system performance, especially for storage drivers that use a copy-on-write filesystem. Write-intensive applications, such as database storage, are impacted by a performance overhead, particularly if pre-existing data exists in the read-only layer. Use Docker volumes for write-intensive data, data that must persist beyond the container's lifespan, and data that must be shared between containers. Refer to the volumes section to learn how to use volumes to persist data and improve performance. https://docs.docker.com/storage/storagedriver/

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>

Images and layers

A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile:

```
# syntax=docker/dockerfile:1

FROM ubuntu:22.04
LABEL org.opencontainers.image.authors="org@example.com"
COPY . /app
RUN make /app
RUN rm -r $HOME/.cache
CMD python /app/app.py
```

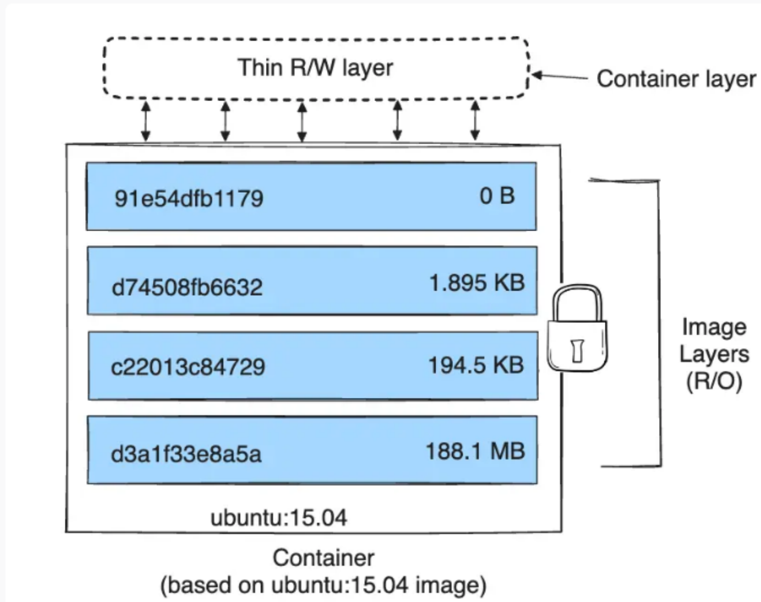
This Dockerfile contains four commands. Commands that modify the filesystem create a layer. The `FROM` statement starts out by creating a layer from the `ubuntu:22.04` image. The `LABEL` command only modifies the image's metadata, and doesn't produce a new layer. The `COPY` command adds some files from your Docker client's current directory. The first `RUN` command builds your application using the `make` command, and writes the result to a new layer. The second `RUN` command removes a cache directory, and writes the result to a new layer. Finally, the `CMD` instruction specifies what command to run within the container, which only modifies the image's metadata, which doesn't produce an image layer.

https://docs.docker.com/storage/storagedriver/

</td></tr>
</table>

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>
Each layer is only a set of differences from the layer before it. Note that both adding, and removing files will result in a new layer. In the example above, the <code>$HOME/.cache</code> directory is removed, but will still be available in the previous layer and add up to the image's total size. Refer to the Best practices for writing Dockerfiles and use multi-stage builds sections to learn how to optimize your Dockerfiles for efficient images.

The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The diagram below shows a container based on an <code>ubuntu:15.04</code> image.



https://docs.docker.com/storage/storagedriver/
</td></tr>
</table>

Claim 1	Accused Instrumentalities
	The original purpose of the cgroup, **chroot**, and namespace facilities in the kernel was to protect applications from noisy, nosey, and messy neighbors. Combining these with container images created an abstraction that also isolates applications from the (heterogeneous) operating systems https://kubernetes.io/docs/concepts/storage/volumes/ **Container environment** The Kubernetes Container environment provides several important resources to Containers: • A filesystem, which is a combination of an image and one or more volumes. • Information about the Container itself. • Information about other objects in the cluster. https://kubernetes.io/docs/concepts/containers/container-environment/

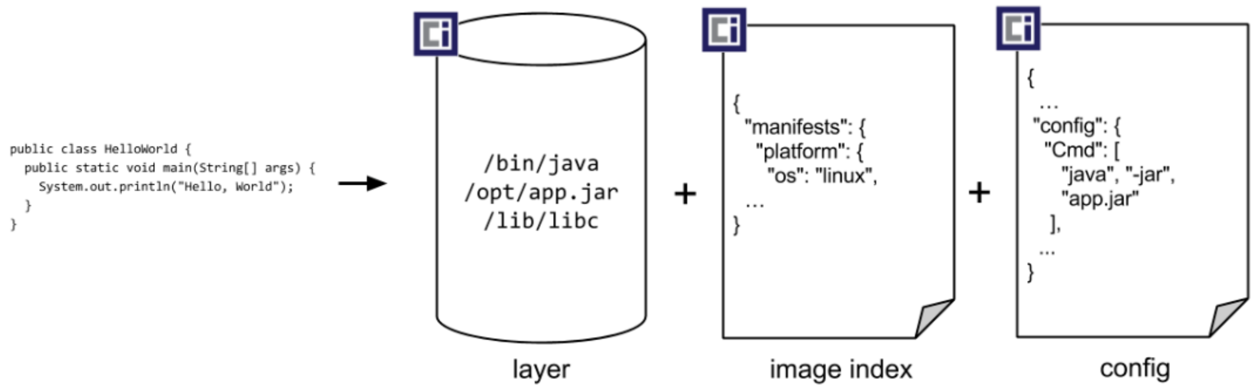
Claim 1	Accused Instrumentalities
	Images A container image represents binary data that encapsulates an application and all its software dependencies. Container images are executable software bundles that can run standalone and that make very well defined assumptions about their runtime environment. You typically create a container image of your application and push it to a registry before referring to it in a Pod. https://kubernetes.io/docs/concepts/containers/images/ **Volumes** On-disk files in a container are ephemeral, which presents some problems for non-trivial applications when running in containers. One problem occurs when a container crashes or is stopped. Container state is not saved so all of the files that were created or modified during the lifetime of the container are lost. During a crash, kubelet restarts the container with a clean state. Another problem occurs when multiple containers are running in a `Pod` and need to share files. It can be challenging to setup and access a shared filesystem across all of the containers. The Kubernetes volume abstraction solves both of these problems. Familiarity with Pods is suggested. https://kubernetes.io/docs/concepts/storage/volumes/

Claim 1	Accused Instrumentalities
	Open Container Initiative **Image Format Specification** This specification defines an OCI Image, consisting of an image manifest, an image index (optional), a set of filesystem layers, and a configuration. The goal of this specification is to enable the creation of interoperable tools for building, transporting, and preparing a container image to run. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>

Overview

At a high level the image manifest contains metadata about the contents and dependencies of the image including the content-addressable identity of one or more filesystem layer changeset archives that will be unpacked to make up the final runnable filesystem. The image configuration includes information such as application arguments, environments, etc. The image index is a higher-level manifest which points to a list of manifests and descriptors. Typically, these manifests may provide different implementations of the image, possibly varying by platform or other attributes.



https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

</td></tr>
</table>

Claim 1	Accused Instrumentalities
	# OCI Image Configuration An OCI *Image* is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime. This specification outlines the JSON format describing images for use with a container runtime and execution tool and its relationship to filesystem changesets, described in Layers. This section defines the `application/vnd.oci.image.config.v1+json` media type. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>
<h2>Layer</h2>

Image filesystems are composed of layers.
Each layer represents a set of filesystem changes in a tar-based layer format, recording files to be added, changed, or deleted relative to its parent layer.
Layers do not have configuration metadata such as environment variables or default arguments - these are properties of the image as a whole rather than any particular layer.
Using a layer-based or union filesystem such as AUFS, or by computing the diff from filesystem snapshots, the filesystem changeset can be used to present a series of image layers as if they were one cohesive filesystem.

<h2>Image JSON</h2>

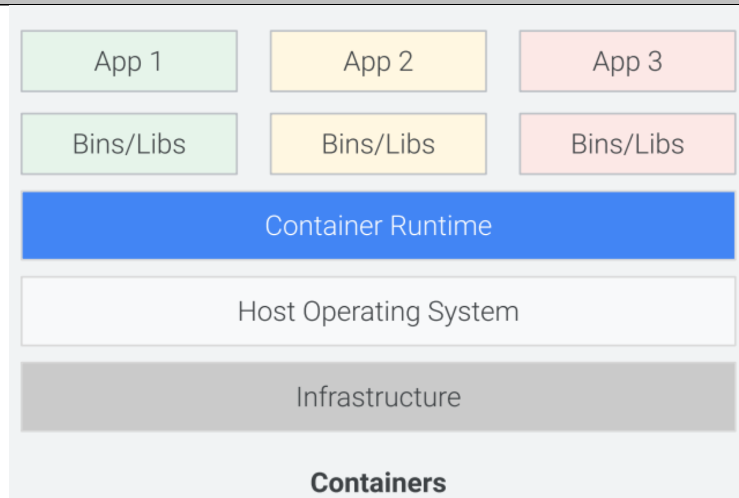
Each image has an associated JSON structure which describes some basic information about the image such as date created, author, as well as execution/runtime configuration like its entrypoint, default arguments, networking, and volumes.
The JSON structure also references a cryptographic hash of each layer used by the image, and provides history information for those layers.
This JSON is considered to be immutable, because changing it would change the computed ImageID.
Changing it means creating a new derived image, instead of changing the existing image.

https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md
</td></tr>
</table>

Claim 1	Accused Instrumentalities
	• **rootfs** *object*, REQUIRED The rootfs key references the layer content addresses used by the image. This makes the image config hash depend on the filesystem hash. ○ **type** *string*, REQUIRED MUST be set to `layers`. Implementations MUST generate an error if they encounter a unknown value while verifying or unpacking an image. ○ **diff_ids** *array of strings*, REQUIRED An array of layer content hashes (`DiffIDs`), in order from first to last. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

Claim 2

Claim 2	Accused Instrumentalities
2. A method as defined in claim 1, wherein each container has an execution file associated therewith for starting the one or more applications.	Google and/or its customer practices, through the Accused Instrumentalities, a method as defined in claim 1, wherein each container has an execution file associated therewith for starting the one or more applications. For example, a container image has an associated image configuration comprising information for starting the one or more applications. This can be an Open Containers Initiative image configuration. *See, e.g.*:

<table>
<tr><th>Claim 2</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>


https://services.google.com/fh/files/misc/why_container_security_matters.pdf

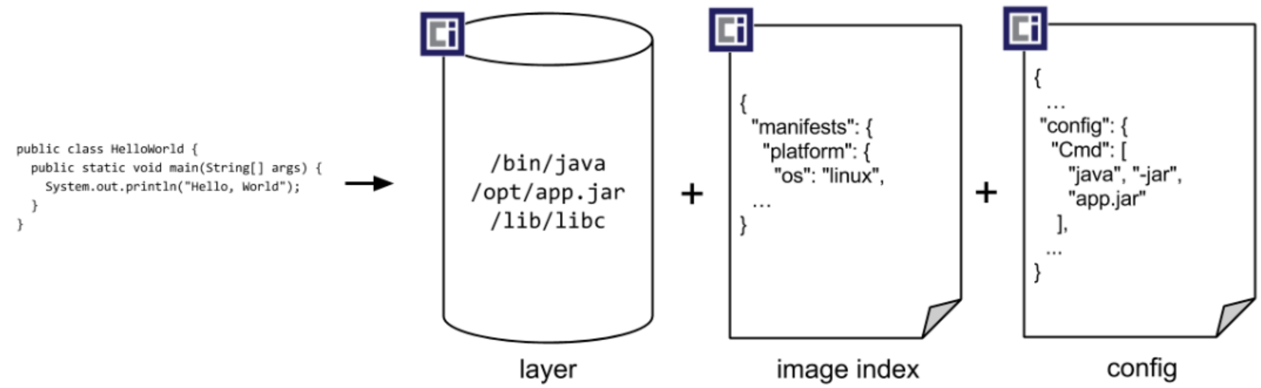
<h1>Open Container Initiative</h1>

<h2>Image Format Specification</h2>

This specification defines an OCI Image, consisting of an image manifest, an image index (optional), a set of filesystem layers, and a configuration.

The goal of this specification is to enable the creation of interoperable tools for building, transporting, and preparing a container image to run.

https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md
</td></tr>
</table>

Claim 2	Accused Instrumentalities
	Overview At a high level the image manifest contains metadata about the contents and dependencies of the image including the content-addressable identity of one or more filesystem layer changeset archives that will be unpacked to make up the final runnable filesystem. The image configuration includes information such as application arguments, environments, etc. The image index is a higher-level manifest which points to a list of manifests and descriptors. Typically, these manifests may provide different implementations of the image, possibly varying by platform or other attributes.  https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

Claim 2	Accused Instrumentalities
	OCI Image Configuration An OCI *Image* is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime. This specification outlines the JSON format describing images for use with a container runtime and execution tool and its relationship to filesystem changesets, described in Layers. This section defines the `application/vnd.oci.image.config.v1+json` media type. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

Claim 2	Accused Instrumentalities
	• **config** *object*, OPTIONAL The execution parameters which SHOULD be used as a base when running a container using the image. This field can be `null`, in which case any execution parameters should be specified at creation of the container. ◦ **Env** *array of strings*, OPTIONAL Entries are in the format of `VARNAME=VARVALUE`. These values act as defaults and are merged with any specified when creating a container. ◦ **Entrypoint** *array of strings*, OPTIONAL A list of arguments to use as the command to execute when the container starts. These values act as defaults and may be replaced by an entrypoint specified when creating a container. ◦ **Cmd** *array of strings*, OPTIONAL Default arguments to the entrypoint of the container. These values act as defaults and may be replaced by any specified when creating a container. If an `Entrypoint` value is not specified, then the first entry of the `Cmd` array SHOULD be interpreted as the executable to run. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

<u>Claim 4</u>

Claim 4	Accused Instrumentalities
4. A method as defined in claim 1 further comprising the step of pre-identifying applications and system files required for association with the one or more containers prior to said storing step.	Google and/or its customer practices, through the Accused Instrumentalities, a method as defined in claim 1 further comprising the step of pre-identifying applications and system files required for association with the one or more containers prior to said storing step. For example, Google's Migrate to Containers feature identifies the application along with its dependencies to be migrated to the target cluster/container. This identification step happens before storing the containers having the migrated application and files in the target machine, as described above. *See* analysis and evidence for claim 1 above. *See also, e.g.*: The migration prerequisites are dependent on your specific migration environment. Confirm that your workloads' OS and source platform are compatible for migration by reviewing the prerequisites for your specific migration environment: https://cloud.google.com/migrate/containers/docs/setting-up-overview Migrate data Send feedback This page describes how to run a data migration that copies files from the local machine to a persistent volume claim (PVC) in the target cluster. https://cloud.google.com/migrate/containers/docs/m2c-cli/migrate-data Copy the source machine's file system Send feedback Modernization of an application component requires creating a copy of the source machine's file system. This page describes the steps required to copy the source machine's file system along with some specifications for reducing the size of the copied file system. https://cloud.google.com/migrate/containers/docs/m2c-cli/copy-file-system

<u>Claim 6</u>

Claim 6	Accused Instrumentalities
6. A method as defined in claim 2, comprising the step of assigning a unique associated identity to each of a plurality of the containers, wherein the identity includes at least one of IP address, host name, and MAC address.	Google and/or its customer practices, through the Accused Instrumentalities, a method as defined in claim 2, comprising the step of assigning a unique associated identity to each of a plurality of the containers, wherein the identity includes at least one of IP address, host name, and MAC address. For example, Kubernetes containers have an associated hostname, which in the case of a single-container Pod is the unique identity of that container. For another example, Kubernetes pods have an associated hostname, which is unique. For another example, a networked Kubernetes pod has an assigned IPv4 and/or IPv6 address. For another example, a Docker container has an IP address and a hostname. *See, e.g.*: **Container information** The *hostname* of a Container is the name of the Pod in which the Container is running. It is available through the `hostname` command or the `gethostname` function call in libc. The Pod name and namespace are available as environment variables through the downward API. User defined environment variables from the Pod definition are also available to the Container, as are any environment variables specified statically in the container image. https://kubernetes.io/docs/concepts/containers/container-environment/

Claim 6	Accused Instrumentalities
	IP address and hostname By default, the container gets an IP address for every Docker network it attaches to. A container receives an IP address out of the IP subnet of the network. The Docker daemon performs dynamic subnetting and IP address allocation for containers. Each network also has a default subnet mask and gateway. You can connect a running container to multiple networks, either by passing the `--network` flag multiple times when creating the container, or using the `docker network connect` command for already running containers. In both cases, you can use the `--ip` or `--ip6` flags to specify the container's IP address on that particular network. In the same way, a container's hostname defaults to be the container's ID in Docker. You can override the hostname using `--hostname`. When connecting to an existing network using `docker network connect`, you can use the `--alias` flag to specify an additional network alias for the container on that network. https://docs.docker.com/network/

Claim 9

Claim 9	Accused Instrumentalities
9. A method as defined in claim 2, wherein server information related to hardware resource usage including at least one of CPU memory, network bandwidth, and disk allocation is associated with at least some of the containers prior to the	Google and/or its customer practices, through the Accused Instrumentalities, a method as defined in claim 2, wherein server information related to hardware resource usage including at least one of CPU memory, network bandwidth, and disk allocation is associated with at least some of the containers prior to the applications within the containers being executed. For example, Kubernetes tracks and limits resource usage, including CPU and memory resources. For another example, Docker tracks and limits resource usage, including CPU and memory resources. *See, e.g.*:

Claim 9	Accused Instrumentalities
applications within the containers being executed.	**Resource Management for Pods and Containers** When you specify a Pod, you can optionally specify how much of each resource a container needs. The most common resources to specify are CPU and memory (RAM); there are others. When you specify the resource *request* for containers in a Pod, the kube-scheduler uses this information to decide which node to place the Pod on. When you specify a resource *limit* for a container, the kubelet enforces those limits so that the running container is not allowed to use more of that resource than the limit you set. The kubelet also reserves at least the *request* amount of that system resource specifically for that container to use. Requests and limits If the node where a Pod is running has enough of a resource available, it's possible (and allowed) for a container to use more resource than its `request` for that resource specifies. However, a container is not allowed to use more than its resource `limit`. For example, if you set a `memory` request of 256 MiB for a container, and that container is in a Pod scheduled to a Node with 8GiB of memory and no other Pods, then the container can try to use more RAM. If you set a `memory` limit of 4GiB for that container, the kubelet (and container runtime) enforce the limit. The runtime prevents the container from using more than the configured resource limit. For example: when a process in the container tries to consume more than the allowed amount of memory, the system kernel

Claim 9	Accused Instrumentalities
	terminates the process that attempted the allocation, with an out of memory (OOM) error. Limits can be implemented either reactively (the system intervenes once it sees a violation) or by enforcement (the system prevents the container from ever exceeding the limit). Different runtimes can have different ways to implement the same restrictions. https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/ **Runtime options with Memory, CPUs, and GPUs** By default, a container has no resource constraints and can use as much of a given resource as the host's kernel scheduler allows. Docker provides ways to control how much memory, or CPU a container can use, setting runtime configuration flags of the `docker run` command. This section provides details on when you should set such limits and the possible implications of setting them. **Limit a container's access to memory** Docker can enforce hard or soft memory limits. • Hard limits lets the container use no more than a fixed amount of memory. • Soft limits lets the container use as much memory as it needs unless certain conditions are met, such as when the kernel detects low memory or contention on the host machine. https://docs.docker.com/config/containers/resource_constraints/

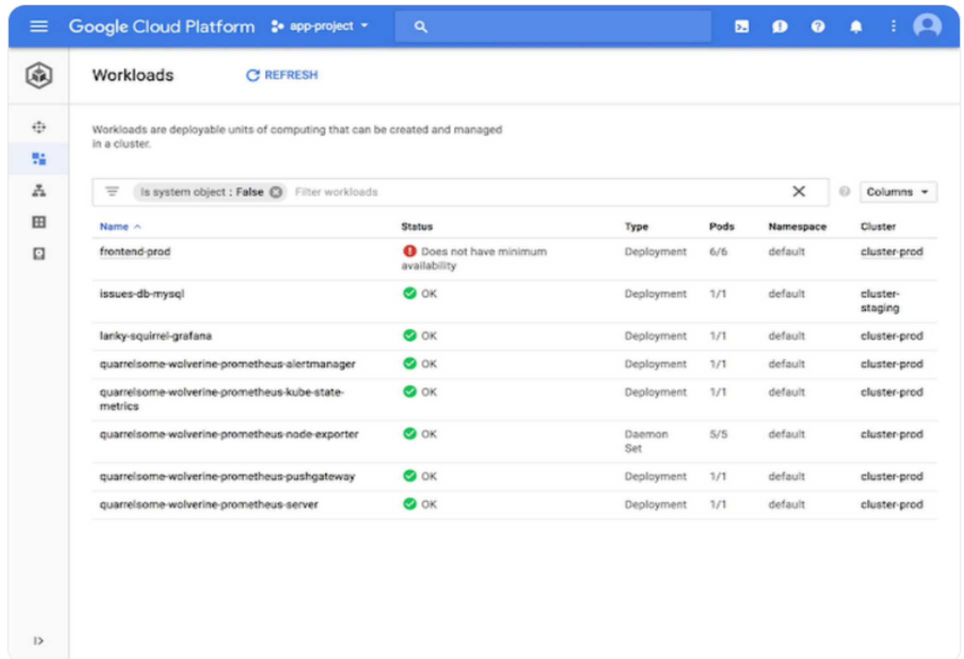
Claim 10

Claim 10	Accused Instrumentalities
10. A method as defined in claim 2, wherein in operation when an application residing within a container is executed, said application has no access to system files or applications in other containers or to system files within the operating system during execution thereof.	Google and/or its customer practices, through the Accused Instrumentalities, a method as defined in claim 2, wherein in operation when an application residing within a container is executed, said application has no access to system files or applications in other containers or to system files within the operating system during execution thereof. *See, e.g.*: Containers solve the portability problem by isolating the application and its dependencies so they can be moved seamlessly between machines. A process running in a container lives isolated from the underlying environment. You control what it can see and what resources it can access. This helps you use resources more efficiently and not worry about the underlying infrastructure. One of the primary reasons to adopt containers is for your applications to be decoupled from the underlying environment and support higher resource utilization by "bin packing" multiple workloads onto each server. As such, the architecture of containers means that they're deployed with multiple containers sharing the same kernel. Containers use primitives of the Linux kernel (cgroups, namespaces) to isolate processes in an environment https://services.google.com/fh/files/misc/why_container_security_matters.pdf

Claim 10	Accused Instrumentalities
	The core components of the Linux kernel that are used for containers are **cgroups** — control groups, which define the resources like CPU and memory which are available to a given process — and **namespaces**, which are a way of separating processes by restricting what each process can see, so that system resources "appear" isolated to the process. https://services.google.com/fh/files/misc/why_container_security_matters.pdf

Claim 13

Claim 13	Accused Instrumentalities
13. A method as defined in claim 1 further comprising the step of associating with a plurality of containers a stored history of when processes related to applications within the container are executed for at least one of, tracking statistics, resource allocation, and for monitoring the status of the application.	Google and/or its customer practices, through the Accused Instrumentalities, a method as defined in claim 1 further comprising the step of associating with a plurality of containers a stored history of when processes related to applications within the container are executed for at least one of, tracking statistics, resource allocation, and for monitoring the status of the application. *See* analysis and evidence for claim 1 above. *See also, e.g.*: • **Logging, monitoring, and tracing**. Capture information on your monitoring, logging, and tracing systems. You can integrate your systems with the Google Cloud Observability, or you can use Google Cloud Observability as your only monitoring, logging, and tracing tool. For example, you can integrate Google Cloud Observability with other services, set up logging interfaces for your preferred programming languages, and use the Cloud Logging agent on your VMs. GKE integrates with Google Cloud Observability and Cloud Audit Logs. You can also customize Cloud Logging logs for GKE with Fluentd and then process logs at scale using Dataflow. https://cloud.google.com/architecture/migrating-containers-kubernetes-gke

Claim 13	Accused Instrumentalities
	 https://cloud.google.com/blog/products/gcp/manage-google-kubernetes-engine-from-cloud-console-dashboard-now-generally-available

Claim 14

Claim 14	Accused Instrumentalities
14. A method as defined in claim 1 comprising the step of creating containers prior to said step of storing containers in memory, wherein containers are created by:	Google and/or its customer practices, through the Accused Instrumentalities, a method as defined in claim 1 comprising the step of creating containers prior to said step of storing containers in memory, wherein containers are created by (a) running an instance of a service on a server; (b) determining which files are being used; and, (c) copying applications and associated system files to memory without overwriting the associated system files so as to provide a second instance of the applications and associated system files.

Claim 14	Accused Instrumentalities
a) running an instance of a service on a server; b) determining which files are being used; and, c) copying applications and associated system files to memory without overwriting the associated system files so as to provide a second instance of the applications and associated system files.	For example, GKE, Cloud Run, and Migrate to Containers support the creation of containers and deploying the containers on the server. The containers are first created and then later deployed/stored on the server. The creation step involves determining which applications and files are to be migrated, copying these identified applications and files to a location in the target server. Based on information and belief, once the files are migrated, the earlier stored files (if any) are not deleted/overwritten, rather, the migrated files are stored as different instance in memory accessible to containers. Further, an instance of an application/service may be tested or run on the target server to ensure compatibility. *See* analysis and evidence for claim 1 above. *See also, e.g.*: **Migrate data** Send feedback This page describes how to run a data migration that copies files from the local machine to a persistent volume claim (PVC) in the target cluster. https://cloud.google.com/migrate/containers/docs/m2c-cli/migrate-data **Copy the source machine's file system** Send feedback Modernization of an application component requires creating a copy of the source machine's file system. This page describes the steps required to copy the source machine's file system along with some specifications for reducing the size of the copied file system. https://cloud.google.com/migrate/containers/docs/m2c-cli/copy-file-system The migration prerequisites are dependent on your specific migration environment. Confirm that your workloads' OS and source platform are compatible for migration by reviewing the prerequisites for your specific migration environment: https://cloud.google.com/migrate/containers/docs/setting-up-overview

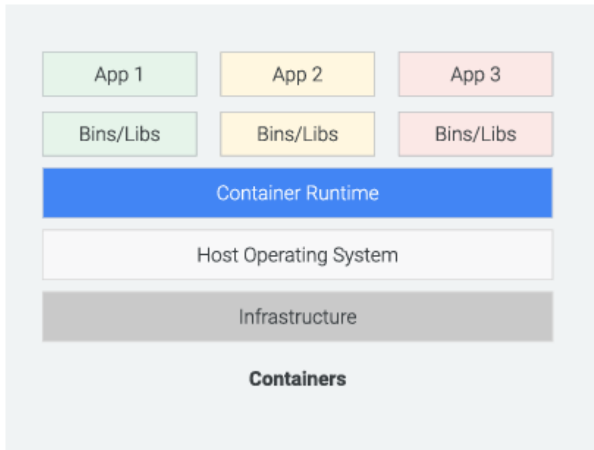
U.S. Patent No. 7,784,058 ("'058 Patent")

Accused Instrumentalities: Google products and services using secure containerized applications, including without limitation Google Kubernetes Engine, Cloud Run, and Migrate to Containers, and all versions and variations thereof since the issuance of the asserted patent.

Each Accused Instrumentality infringes the claims in substantially the same way, and the evidence shown in this chart is similarly applicable to each Accused Instrumentality. Each claim limitation is literally infringed by each Accused Instrumentality. However, to the extent any claim limitation is not met literally, it is nonetheless met under the doctrine of equivalents because the differences between the claim limitation and each Accused Instrumentality would be insubstantial, and each Accused Instrumentality performs substantially the same function, in substantially the same way, to achieve the same result as the claimed invention. Notably, Defendant has not yet articulated which, if any, particular claim limitations it believes are not met by the Accused Instrumentalities.

Claim 1

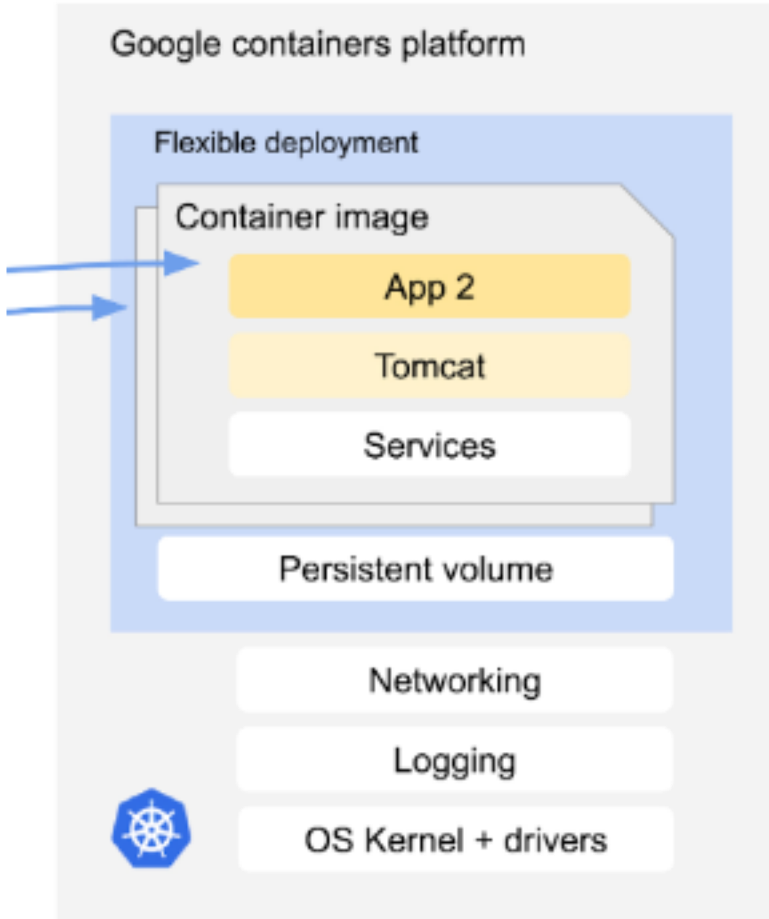
Claim 1	Accused Instrumentalities
[1pre] 1. A computing system for executing a plurality of software applications comprising:	To the extent the preamble is limiting, each Accused Instrumentality comprises or constitutes a computing system for executing a plurality of software applications as claimed. *See* claim limitations below. *See also, e.g.*: Google Kubernetes Engine (GKE) clusters provide secured and managed Kubernetes services with autoscaling and multi-cluster support. GKE lets you deploy, manage, and scale containerized applications on Kubernetes, powered by Google Cloud. https://cloud.google.com/migrate/containers/docs/getting-started Use Migrate to Containers to modernize traditional applications away from virtual machine (VM) instances and into native containers that run on Google Kubernetes Engine (GKE), GKE Enterprise clusters, or Cloud Run platform. You can migrate workloads from VMs that run on VMware or Compute Engine, giving you the flexibility to containerize your existing workloads with ease. Migrate to Containers supports modernization of IBM WebSphere, JBoss, Apache, Tomcat, WordPress, Windows IIS applications, as well as containerisation of Linux-based applications. https://cloud.google.com/migrate/containers/docs/getting-started.

Claim 1	Accused Instrumentalities
	A container is a way of packaging a given application's code and dependencies so that the application will run easily in any computing environment. This solves the common problem of  https://services.google.com/fh/files/misc/why_container_security_matters.pdf

Claim 1	Accused Instrumentalities
	Containers can run virtually anywhere, greatly easing development and deployment: on Linux, Windows, and Mac operating systems; on virtual machines or on physical servers; on a developer's machine or in data centers on-premises; and of course, in the public cloud. Containers are lightweight packages of your application code together with dependencies such as specific versions of programming language runtimes and libraries required to run your software services. https://cloud.google.com/learn/what-are-containers
[1a] a) a processor;	Each Accused Instrumentality comprises a processor. *See, e.g.*:

Claim 1	Accused Instrumentalities
	Containers virtualize CPU, memory, storage, and network resources at the operating system level, providing developers with a view of the OS logically isolated from other applications. https://cloud.google.com/learn/what-are-containers • **Higher utilization and density**, leveraging automatic bin-packing and auto-scaling capabilities, Kubernetes places containers optimally in nodes based on required resources while scaling as needed, without impairing availability. In addition, unlike VMs, all containers on a single node share one copy of the operating system and don't each require their own OS image and vCPU, resulting in a much smaller memory footprint and CPU needs. This means more workloads running on fewer compute resources. https://cloud.google.com/blog/products/containers-kubernetes/how-migrate-for-anthos-improves-vm-to-container-migration Containers use specific features of the Linux kernel that "trick" individual applications into thinking they're in their own unique environment, even though multiple applications share the same host kernel. (If you're not familiar with the Linux kernel, it's a part of the operating system that communicates between processes--requests that do user tasks like opening a file, running a program-- and the hardware. It manages resources like memory and CPU to meet these requests). https://services.google.com/fh/files/misc/why_container_security_matters.pdf

Claim 1	Accused Instrumentalities
[1b] b) an operating system having an operating system kernel having OS critical system elements (OSCSEs) for running in kernel mode using said processor; and,	Each Accused Instrumentality comprises an operating system having an operating system kernel having OS critical system elements (OSCSEs) for running in kernel mode using said processor. *See, e.g.*: • Containers are much more lightweight than VMs • Containers virtualize at the OS level while VMs virtualize at the hardware level • Containers share the OS kernel and use a fraction of the memory VMs require https://cloud.google.com/learn/what-are-containers **Kernel mode** Kernel mode refers to the processor mode that enables software to have full and unrestricted access to the system and its resources. The OS kernel and kernel drivers, such as the file system driver, are loaded into protected memory space and operate in this highly privileged kernel mode. https://www.techtarget.com/searchdatacenter/definition/kernel

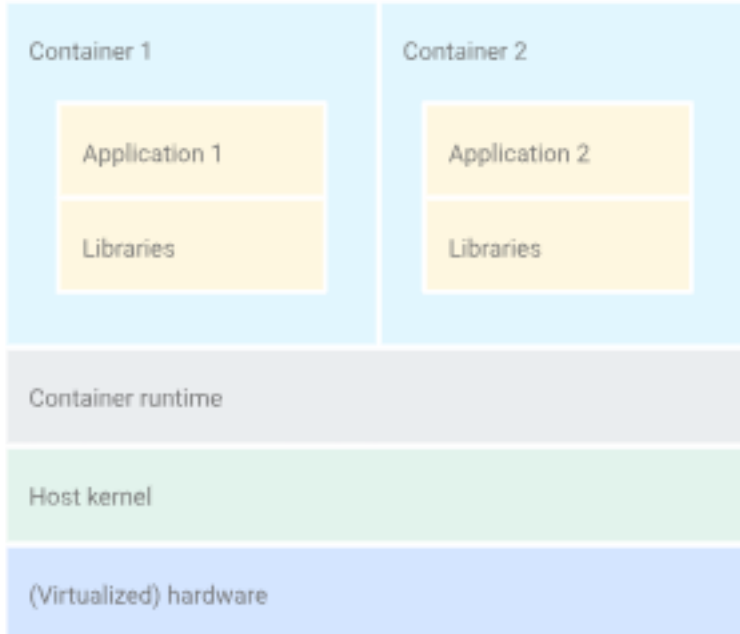
Claim 1	Accused Instrumentalities
	 https://cloud.google.com/blog/products/application-modernization/shift-your-apps-to-container-based-workloads-on-the-command-line

Claim 1	Accused Instrumentalities
	The migration prerequisites are dependent on your specific migration environment. Confirm that your workloads' OS and source platform are compatible for migration by reviewing the prerequisites for your specific migration environment: https://cloud.google.com/migrate/containers/docs/setting-up-overview Containers use specific features of the Linux kernel that "trick" individual applications into thinking they're in their own unique environment, even though multiple applications share the same host kernel. (If you're not familiar with the Linux kernel, it's a part of the operating system that communicates between processes--requests that do user tasks like opening a file, running a program-- and the hardware. It manages resources like memory and CPU to meet these requests). https://services.google.com/fh/files/misc/why_container_security_matters.pdf The **GNU C Library**, commonly known as **glibc**, is the GNU Project implementation of the C standard library. It is a wrapper around the system calls of the Linux kernel for application use. Despite its name, it now also directly supports C++ (and, indirectly, other programming languages). It was started in the 1980s by the Free Software Foundation (FSF) for the GNU operating system. https://en.wikipedia.org/wiki/Glibc
[1c] c) a shared library having shared library critical system elements (SLCSEs) stored therein for use by the plurality of software applications in user mode and	Each Accused Instrumentality comprises a shared library having shared library critical system elements (SLCSEs) stored therein for use by the plurality of software applications in user mode. *See, e.g.*:

Claim 1	Accused Instrumentalities
	A "container image" is your application and its dependencies, and uses a "base image" as the basis for the container image The container image specifies the container's file system. For example, if you're running a Node.js application, the container image would contain your app, Node.js, and other dependencies like Linux system libraries (except the kernel). A container image usually extends a base operating system image, or **base image**. This base image is the basis of your container, so you'll want to ensure that it's properly patched and free from known vulnerabilities. https://services.google.com/fh/files/misc/why_container_security_matters.pdf

Claim 1	Accused Instrumentalities
	A base image is the starting point for most container-based development workflows. Developers start with a base image and layer on top of it the necessary libraries, binaries, and configuration files used to run their application. Many base images are basic or minimal Linux distributions: Debian, Ubuntu, Red Hat Enterprise Linux (RHEL), Rocky Linux, or Alpine. Developers can consume these images directly from Docker Hub or other sources. There are official providers along with a wide variety of other downstream repackagers that layer software to meet customer needs. Google maintains base images for building its own applications. These images are built from the same source that Docker Hub uses. Therefore, they match the images you would get from Docker Hub. https://cloud.google.com/software-supply-chain-security/docs/base-images The preconfigured base images provided by Cloud Workstations contain only a minimal environment with IDE, basic Linux terminal and language tools and a `sshd` server. To expedite the environment setup of specific development use cases, you can create custom container images that extend these base images to pre-install tools and dependencies and that run automation scripts. For custom container images, we recommend setting up a pipeline to automatically rebuild these images when the Cloud Workstations base image is updated, in addition to running a container scanning tool such as Artifact Analysis to inspect any additional dependencies you added. You're responsible for maintaining and updating custom packages and dependencies added to custom images. https://cloud.google.com/workstations/docs/customize-container-images

Claim 1	Accused Instrumentalities
	A container is a way of packaging a given application's code and dependencies so that the application will run easily in any computing environment. This solves the common problem of Containers solve the portability problem by isolating the application and its dependencies so they can be moved seamlessly between machines. A process running in a container lives isolated from the underlying environment. You control what it can see and what resources it can access. This helps you use resources more efficiently and not worry about the underlying infrastructure. One of the primary reasons to adopt containers is for your applications to be decoupled from the underlying environment and support higher resource utilization by "bin packing" multiple workloads onto each server. As such, the architecture of containers means that they're deployed with multiple containers sharing the same kernel. The core components of the Linux kernel that are used for containers are **cgroups** — control groups, which define the resources like CPU and memory which are available to a given process — and **namespaces**, which are a way of separating processes by restricting what each process can see, so that system resources "appear" isolated to the process. https://services.google.com/fh/files/misc/why_container_security_matters.pdf

Claim 1	Accused Instrumentalities
	 https://cloud.google.com/architecture/best-practices-for-operating-containers For example, Migrate to Containers automatically generates a container image, a Dockerfile for day-2 image updates and application revisions, Kubernetes deployment YAMLs and (where relevant) a persistent data volume onto which the application data files and persistent state are copied. This automated, intelligent extraction is https://cloud.google.com/blog/products/containers-kubernetes/how-migrate-for-anthos-improves-vm-to-container-migration

Claim 1	Accused Instrumentalities
	Containers are lightweight packages of your application code together with dependencies such as specific versions of programming language runtimes and libraries required to run your software services. https://cloud.google.com/learn/what-are-containers # About storage drivers To use storage drivers effectively, it's important to know how Docker builds and stores images, and how these images are used by containers. You can use this information to make informed choices about the best way to persist data from your applications and avoid performance problems along the way. ## Storage drivers versus Docker volumes Docker uses storage drivers to store image layers, and to store data in the writable layer of a container. The container's writable layer doesn't persist after the container is deleted, but is suitable for storing ephemeral data that is generated at runtime. Storage drivers are optimized for space efficiency, but (depending on the storage driver) write speeds are lower than native file system performance, especially for storage drivers that use a copy-on-write filesystem. Write-intensive applications, such as database storage, are impacted by a performance overhead, particularly if pre-existing data exists in the read-only layer. Use Docker volumes for write-intensive data, data that must persist beyond the container's lifespan, and data that must be shared between containers. Refer to the volumes section to learn how to use volumes to persist data and improve performance. https://docs.docker.com/storage/storagedriver/

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>

Images and layers

A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile:

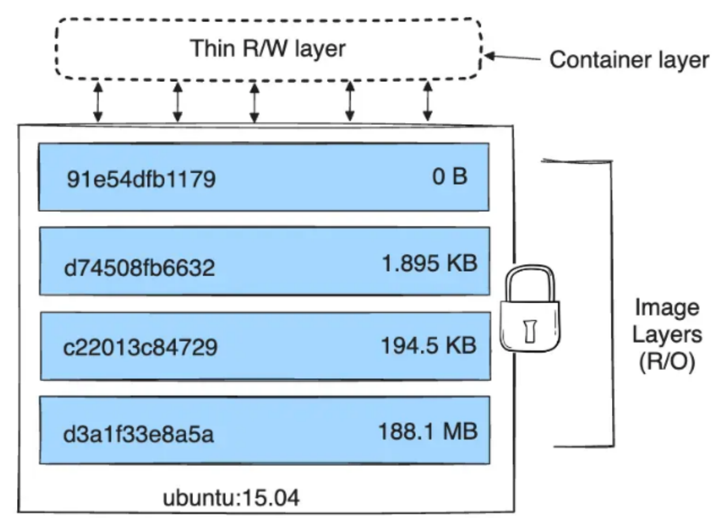
```
# syntax=docker/dockerfile:1

FROM ubuntu:22.04
LABEL org.opencontainers.image.authors="org@example.com"
COPY . /app
RUN make /app
RUN rm -r $HOME/.cache
CMD python /app/app.py
```

This Dockerfile contains four commands. Commands that modify the filesystem create a layer. The `FROM` statement starts out by creating a layer from the `ubuntu:22.04` image. The `LABEL` command only modifies the image's metadata, and doesn't produce a new layer. The `COPY` command adds some files from your Docker client's current directory. The first `RUN` command builds your application using the `make` command, and writes the result to a new layer. The second `RUN` command removes a cache directory, and writes the result to a new layer. Finally, the `CMD` instruction specifies what command to run within the container, which only modifies the image's metadata, which doesn't produce an image layer.

https://docs.docker.com/storage/storagedriver/

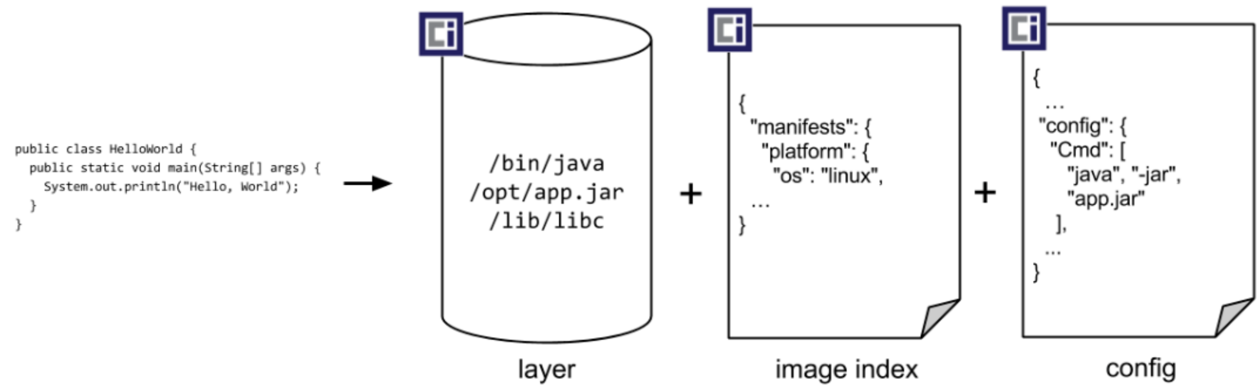
</td></tr>
</table>

Claim 1	Accused Instrumentalities
	Each layer is only a set of differences from the layer before it. Note that both *adding*, and *removing* files will result in a new layer. In the example above, the `$HOME/.cache` directory is removed, but will still be available in the previous layer and add up to the image's total size. Refer to the Best practices for writing Dockerfiles and use multi-stage builds sections to learn how to optimize your Dockerfiles for efficient images. The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The diagram below shows a container based on an `ubuntu:15.04` image.  https://docs.docker.com/storage/storagedriver/

Claim 1	Accused Instrumentalities
	Volumes Volumes are the preferred mechanism for persisting data generated by and used by Docker containers. While bind mounts are dependent on the directory structure and OS of the host machine, volumes are completely managed by Docker. Volumes have several advantages over bind mounts: https://kubernetes.io/docs/concepts/storage/volumes/ **Container environment** The Kubernetes Container environment provides several important resources to Containers: • A filesystem, which is a combination of an image and one or more volumes. • Information about the Container itself. • Information about other objects in the cluster. https://kubernetes.io/docs/concepts/containers/container-environment/

Claim 1	Accused Instrumentalities
	Images A container image represents binary data that encapsulates an application and all its software dependencies. Container images are executable software bundles that can run standalone and that make very well defined assumptions about their runtime environment. You typically create a container image of your application and push it to a registry before referring to it in a Pod. https://kubernetes.io/docs/concepts/containers/images/ **Volumes** On-disk files in a container are ephemeral, which presents some problems for non-trivial applications when running in containers. One problem occurs when a container crashes or is stopped. Container state is not saved so all of the files that were created or modified during the lifetime of the container are lost. During a crash, kubelet restarts the container with a clean state. Another problem occurs when multiple containers are running in a `Pod` and need to share files. It can be challenging to setup and access a shared filesystem across all of the containers. The Kubernetes volume abstraction solves both of these problems. Familiarity with Pods is suggested. https://kubernetes.io/docs/concepts/storage/volumes/

Claim 1	Accused Instrumentalities
	Open Container Initiative **Image Format Specification** This specification defines an OCI Image, consisting of an image manifest, an image index (optional), a set of filesystem layers, and a configuration. The goal of this specification is to enable the creation of interoperable tools for building, transporting, and preparing a container image to run. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

Claim 1	Accused Instrumentalities
	Overview At a high level the image manifest contains metadata about the contents and dependencies of the image including the content-addressable identity of one or more filesystem layer changeset archives that will be unpacked to make up the final runnable filesystem. The image configuration includes information such as application arguments, environments, etc. The image index is a higher-level manifest which points to a list of manifests and descriptors. Typically, these manifests may provide different implementations of the image, possibly varying by platform or other attributes.  https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

Claim 1	Accused Instrumentalities
	OCI Image Configuration An OCI *Image* is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime. This specification outlines the JSON format describing images for use with a container runtime and execution tool and its relationship to filesystem changesets, described in Layers. This section defines the `application/vnd.oci.image.config.v1+json` media type. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>
<h2>Layer</h2>

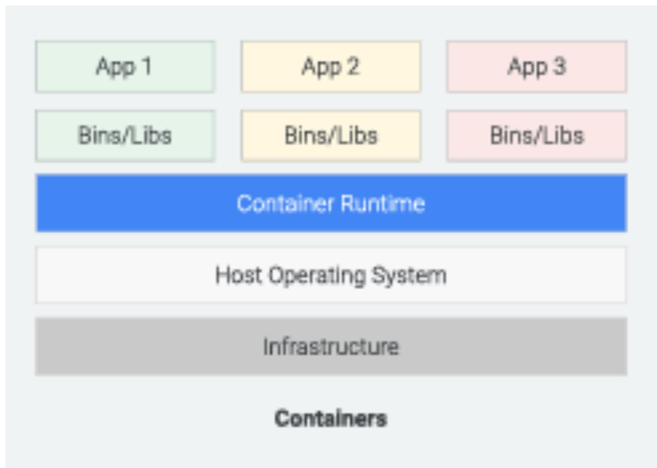
Image filesystems are composed of layers.
Each layer represents a set of filesystem changes in a tar-based layer format, recording files to be added, changed, or deleted relative to its parent layer.
Layers do not have configuration metadata such as environment variables or default arguments - these are properties of the image as a whole rather than any particular layer.
Using a layer-based or union filesystem such as AUFS, or by computing the diff from filesystem snapshots, the filesystem changeset can be used to present a series of image layers as if they were one cohesive filesystem.

<h2>Image JSON</h2>

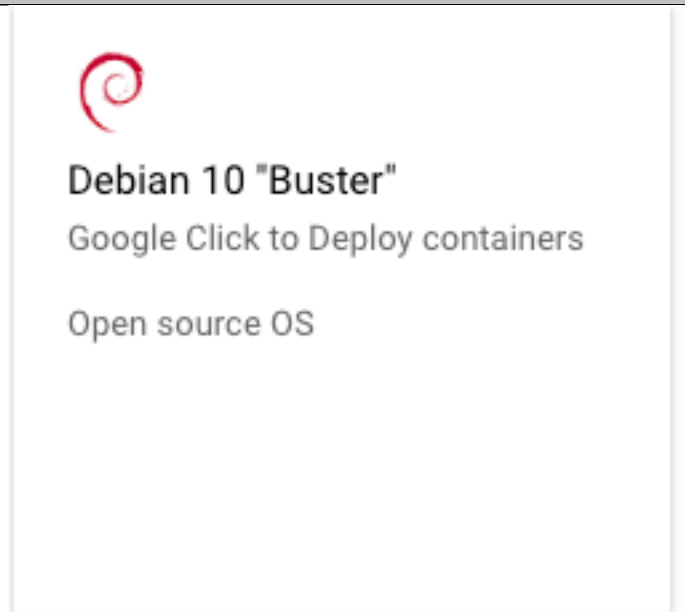
Each image has an associated JSON structure which describes some basic information about the image such as date created, author, as well as execution/runtime configuration like its entrypoint, default arguments, networking, and volumes.
The JSON structure also references a cryptographic hash of each layer used by the image, and provides history information for those layers.
This JSON is considered to be immutable, because changing it would change the computed ImageID.
Changing it means creating a new derived image, instead of changing the existing image.

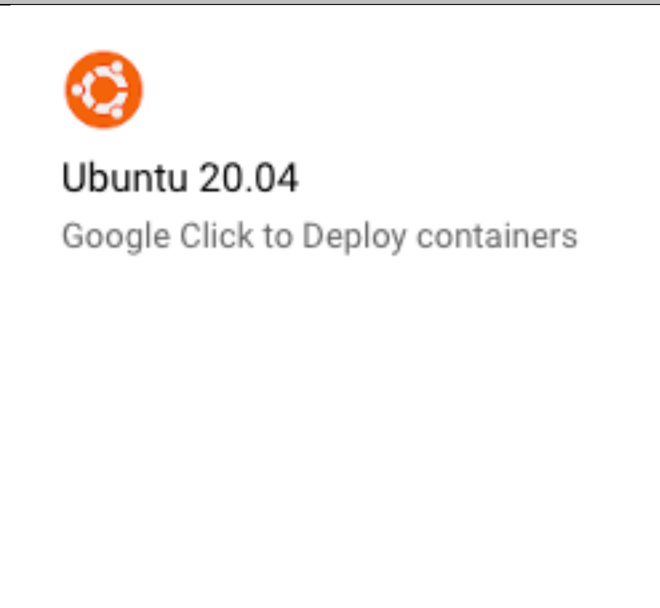
https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md
</td></tr>
<tr><td>[1d] i) wherein some of the SLCSEs stored in the shared library are functional replicas of OSCSEs and are accessible</td><td>In each Accused Instrumentality, some of the SLCSEs stored in the shared library are functional replicas of OSCSEs and are accessible to some of the plurality of software applications and when</td></tr>
</table>

Claim 1	Accused Instrumentalities
to some of the plurality of software applications and when one of the SLCSEs is accessed by one or more of the plurality of software applications it forms a part of the one or more of the plurality of software applications,	one of the SLCSEs is accessed by one or more of the plurality of software applications it forms a part of the one or more of the plurality of software applications. For example, a base image serves as a self-contained unit that encompasses all the necessary components for an application to run, including the application code, runtime environment, system tools, and dependencies (i.e., SLCSEs). The images are based on existing Linux distributions, such as Debian and Ubuntu, including essential system elements (i.e., functional replicas of OSCSEs). Each container image is based on a specific base image, which contains the application code, and dependencies, including system libraries or shared library critical system elements (SLCSEs). When the container runs the image, it creates a runtime instance of that container image. *See, e.g.*: Many base images are basic or minimal Linux distributions: Debian, Ubuntu, Red Hat Enterprise Linux (RHEL), Rocky Linux, or Alpine. Developers can consume these images directly from Docker Hub or other sources. There are official providers along with a wide variety of other downstream repackagers that layer software to meet customer needs. https://cloud.google.com/software-supply-chain-security/docs/base-images A container is a way of packaging a given application's code and dependencies so that the application will run easily in any computing environment. This solves the common problem of

Claim 1	Accused Instrumentalities
	A “container image” is your application and its dependencies, and uses a “base image” as the basis for the container image 

Claim 1	Accused Instrumentalities
	The container image specifies the container's file system. For example, if you're running a Node.js application, the container image would contain your app, Node.js, and other dependencies like Linux system libraries (except the kernel). A container image usually extends a base operating system image, or **base image**. This base image is the basis of your container, so you'll want to ensure that it's properly patched and free from known vulnerabilities. https://services.google.com/fh/files/misc/why_container_security_matters.pdf OS | Repository path | Google Cloud Marketplace listing Debian 10 "Buster" | `marketplace.gcr.io/google/debian10` | Google Cloud Marketplace Debian 11 "Bullseye" | `marketplace.gcr.io/google/debian11` | Google Cloud Marketplace Debian 12 "Bookworm" | `marketplace.gcr.io/google/debian12` | Google Cloud Marketplace Rocky Linux 8 | `marketplace.gcr.io/google/rockylinux8` | Google Cloud Marketplace Rocky Linux 9 | `marketplace.gcr.io/google/rockylinux9` | Google Cloud Marketplace Ubuntu 20.04 | `marketplace.gcr.io/google/ubuntu2004` | Google Cloud Marketplace Ubuntu 22.04 | `marketplace.gcr.io/google/ubuntu2204` | Google Cloud Marketplace https://cloud.google.com/software-supply-chain-security/docs/base-images

Claim 1	Accused Instrumentalities
	

Claim 1	Accused Instrumentalities
	 https://console.cloud.google.com/marketplace/browse?filter=solution-type:container

Claim 1	Accused Instrumentalities
	# About storage drivers To use storage drivers effectively, it's important to know how Docker builds and stores images, and how these images are used by containers. You can use this information to make informed choices about the best way to persist data from your applications and avoid performance problems along the way. ## Storage drivers versus Docker volumes Docker uses storage drivers to store image layers, and to store data in the writable layer of a container. The container's writable layer doesn't persist after the container is deleted, but is suitable for storing ephemeral data that is generated at runtime. Storage drivers are optimized for space efficiency, but (depending on the storage driver) write speeds are lower than native file system performance, especially for storage drivers that use a copy-on-write filesystem. Write-intensive applications, such as database storage, are impacted by a performance overhead, particularly if pre-existing data exists in the read-only layer. Use Docker volumes for write-intensive data, data that must persist beyond the container's lifespan, and data that must be shared between containers. Refer to the volumes section to learn how to use volumes to persist data and improve performance. https://docs.docker.com/storage/storagedriver/

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>

Images and layers

A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile:

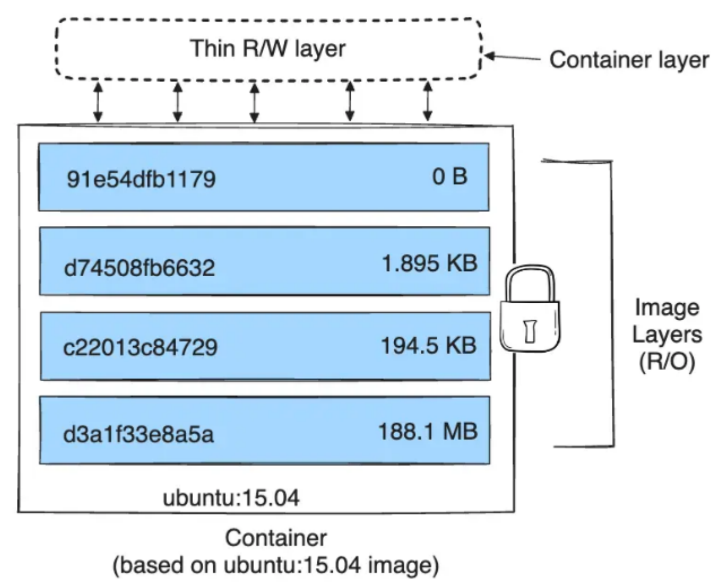
```
# syntax=docker/dockerfile:1

FROM ubuntu:22.04
LABEL org.opencontainers.image.authors="org@example.com"
COPY . /app
RUN make /app
RUN rm -r $HOME/.cache
CMD python /app/app.py
```

This Dockerfile contains four commands. Commands that modify the filesystem create a layer. The `FROM` statement starts out by creating a layer from the `ubuntu:22.04` image. The `LABEL` command only modifies the image's metadata, and doesn't produce a new layer. The `COPY` command adds some files from your Docker client's current directory. The first `RUN` command builds your application using the `make` command, and writes the result to a new layer. The second `RUN` command removes a cache directory, and writes the result to a new layer. Finally, the `CMD` instruction specifies what command to run within the container, which only modifies the image's metadata, which doesn't produce an image layer.

https://docs.docker.com/storage/storagedriver/

</td></tr>
</table>

Claim 1	Accused Instrumentalities
	Each layer is only a set of differences from the layer before it. Note that both *adding*, and *removing* files will result in a new layer. In the example above, the `$HOME/.cache` directory is removed, but will still be available in the previous layer and add up to the image's total size. Refer to the Best practices for writing Dockerfiles and use multi-stage builds sections to learn how to optimize your Dockerfiles for efficient images. The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The diagram below shows a container based on an `ubuntu:15.04` image.  https://docs.docker.com/storage/storagedriver/

Claim 1	Accused Instrumentalities
	# Volumes Volumes are the preferred mechanism for persisting data generated by and used by Docker containers. While bind mounts are dependent on the directory structure and OS of the host machine, volumes are completely managed by Docker. Volumes have several advantages over bind mounts: https://kubernetes.io/docs/concepts/storage/volumes/ ## Container environment The Kubernetes Container environment provides several important resources to Containers: • A filesystem, which is a combination of an image and one or more volumes. • Information about the Container itself. • Information about other objects in the cluster. https://kubernetes.io/docs/concepts/containers/container-environment/

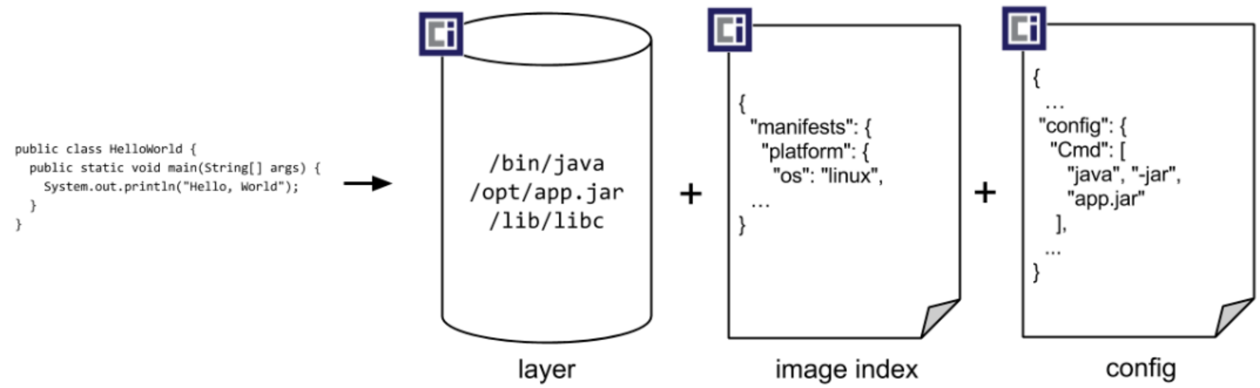
Claim 1	Accused Instrumentalities
	# Images A container image represents binary data that encapsulates an application and all its software dependencies. Container images are executable software bundles that can run standalone and that make very well defined assumptions about their runtime environment. You typically create a container image of your application and push it to a registry before referring to it in a Pod. https://kubernetes.io/docs/concepts/containers/images/ # Volumes On-disk files in a container are ephemeral, which presents some problems for non-trivial applications when running in containers. One problem occurs when a container crashes or is stopped. Container state is not saved so all of the files that were created or modified during the lifetime of the container are lost. During a crash, kubelet restarts the container with a clean state. Another problem occurs when multiple containers are running in a `Pod` and need to share files. It can be challenging to setup and access a shared filesystem across all of the containers. The Kubernetes volume abstraction solves both of these problems. Familiarity with Pods is suggested. https://kubernetes.io/docs/concepts/storage/volumes/

Claim 1	Accused Instrumentalities
	# Open Container Initiative ## Image Format Specification This specification defines an OCI Image, consisting of an image manifest, an image index (optional), a set of filesystem layers, and a configuration. The goal of this specification is to enable the creation of interoperable tools for building, transporting, and preparing a container image to run. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>

Overview

At a high level the image manifest contains metadata about the contents and dependencies of the image including the content-addressable identity of one or more filesystem layer changeset archives that will be unpacked to make up the final runnable filesystem. The image configuration includes information such as application arguments, environments, etc. The image index is a higher-level manifest which points to a list of manifests and descriptors. Typically, these manifests may provide different implementations of the image, possibly varying by platform or other attributes.



https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md
</td></tr>
</table>

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>

OCI Image Configuration

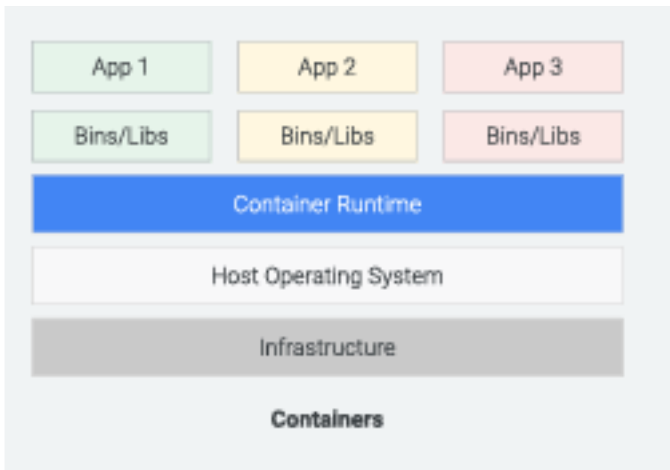
An OCI *Image* is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime. This specification outlines the JSON format describing images for use with a container runtime and execution tool and its relationship to filesystem changesets, described in Layers.

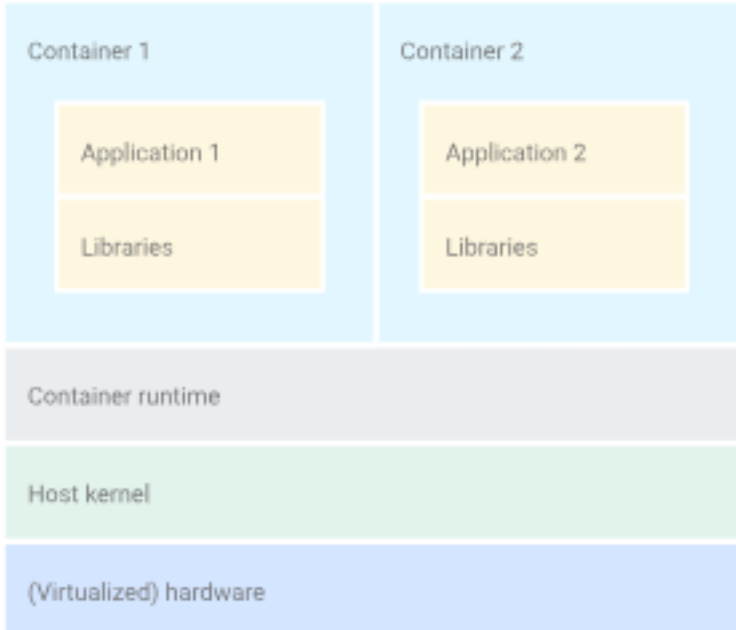
This section defines the `application/vnd.oci.image.config.v1+json` media type.

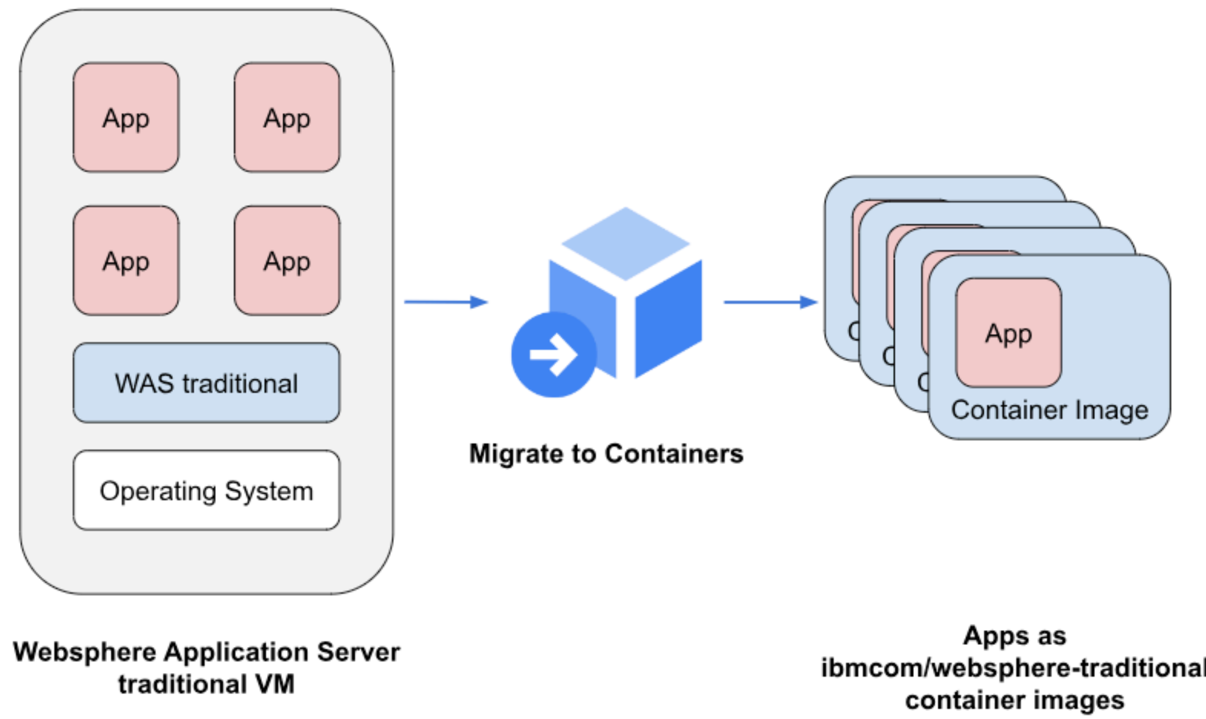
https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md
</td></tr>
</table>

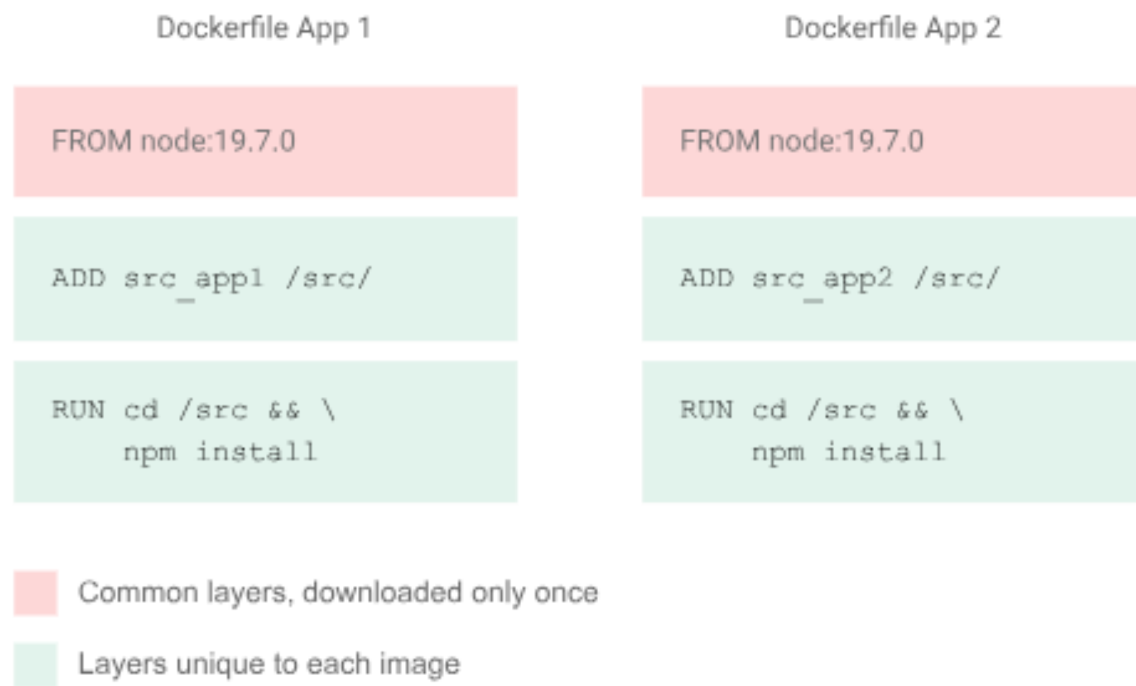
Claim 1	Accused Instrumentalities
	Layer • Image filesystems are composed of *layers*. • Each layer represents a set of filesystem changes in a tar-based layer format, recording files to be added, changed, or deleted relative to its parent layer. • Layers do not have configuration metadata such as environment variables or default arguments - these are properties of the image as a whole rather than any particular layer. • Using a layer-based or union filesystem such as AUFS, or by computing the diff from filesystem snapshots, the filesystem changeset can be used to present a series of image layers as if they were one cohesive filesystem. **Image JSON** • Each image has an associated JSON structure which describes some basic information about the image such as date created, author, as well as execution/runtime configuration like its entrypoint, default arguments, networking, and volumes. • The JSON structure also references a cryptographic hash of each layer used by the image, and provides history information for those layers. • This JSON is considered to be immutable, because changing it would change the computed ImageID. • Changing it means creating a new derived image, instead of changing the existing image. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md
[1e] ii) wherein an instance of a SLCSE provided to at least a first of the plurality of software applications from the	In each Accused Instrumentality, an instance of a SLCSE provided to at least a first of the plurality of software applications from the shared library is run in a context of said at least first of the plurality of software applications without being shared with other of the plurality of software applications and where at least a second of the plurality of software applications running under the

Claim 1	Accused Instrumentalities
shared library is run in a context of said at least first of the plurality of software applications without being shared with other of the plurality of software applications and where at least a second of the plurality of software applications running under the operating system have use of a unique instance of a corresponding critical system element for performing same function, and	operating system have use of a unique instance of a corresponding critical system element for performing same function. When a Docker or Kubernetes image is used to create a container, it creates a separate and isolated instance of a runtime environment which is independent of other containers running on the same host. Each container has its own instance of base images and its own data. The containers run in isolation, ensuring that the SLCSEs stored in the shared library are accessible to the software applications running in their respective containers. The image includes essential system files, libraries, and dependencies required to run the software application within the container. The containers can share common dependencies and components using layered images. This means that multiple containers utilize the same base image to create an instance. When an instance of SLCSE is provided from the base image (i.e., from the shared library) to an individual container including application software, it operates in isolation and runs its own instance of the software application without sharing resources or critical system elements with other containers. This ensures that each container has its own isolated context. Docker or Kubernetes containers can share common dependencies and components using layered images. This means that multiple containers can utilize the same base image. Therefore, each container, containing the application software running under the operating system, utilizes a unique instance of the corresponding critical system element to execute the respective application software for performing a same or a different function. *See, e.g.*: A container is a way of packaging a given application's code and dependencies so that the application will run easily in any computing environment. This solves the common problem of https://services.google.com/fh/files/misc/why_container_security_matters.pdf

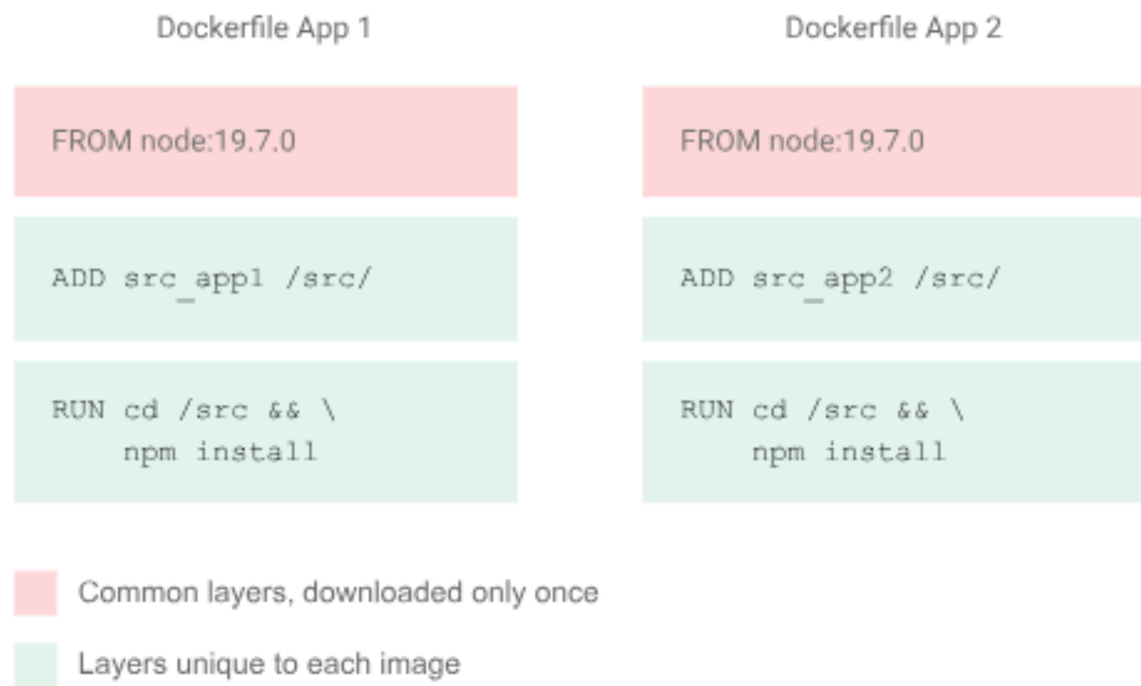
Claim 1	Accused Instrumentalities
	Containers solve the portability problem by isolating the application and its dependencies so they can be moved seamlessly between machines. A process running in a container lives isolated from the underlying environment. You control what it can see and what resources it can access. This helps you use resources more efficiently and not worry about the underlying infrastructure. The container image specifies the container's file system. For example, if you're running a Node.js application, the container image would contain your app, Node.js, and other dependencies like Linux system libraries (except the kernel). A container image usually extends a base operating system image, or **base image**. This base image is the basis of your container, so you'll want to ensure that it's properly patched and free from known vulnerabilities.  https://services.google.com/fh/files/misc/why_container_security_matters.pdf

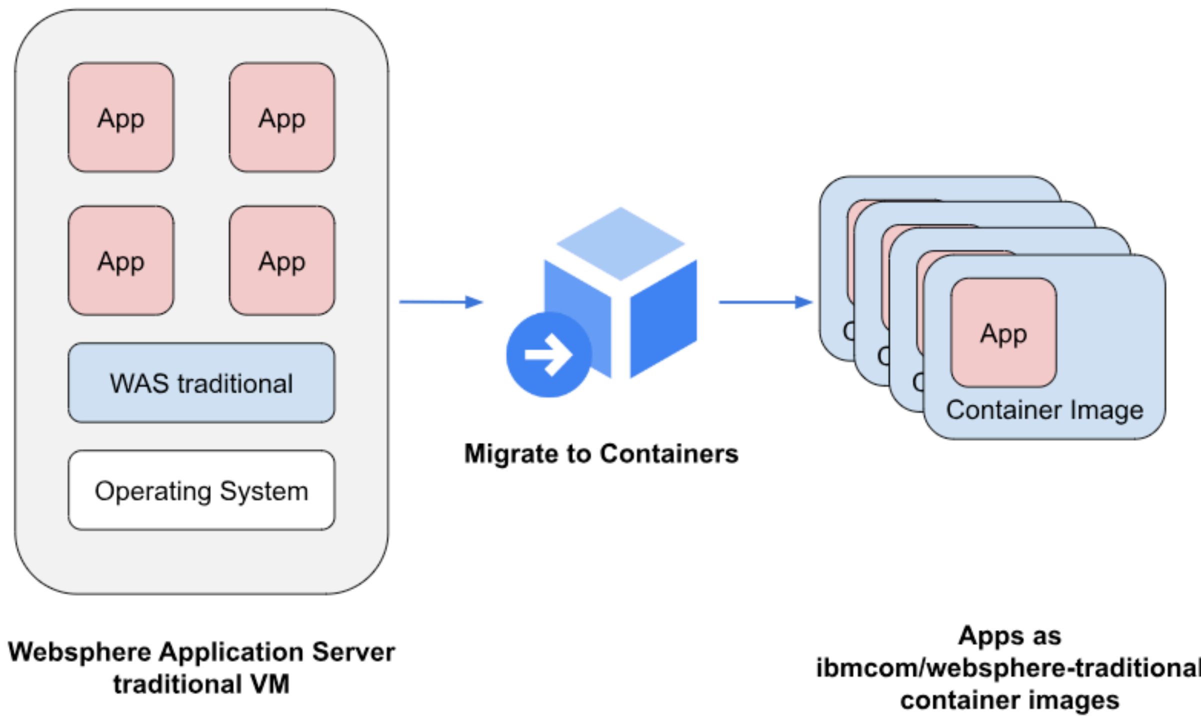
Claim 1	Accused Instrumentalities
	 https://cloud.google.com/architecture/best-practices-for-operating-containers

Claim 1	Accused Instrumentalities
	 https://cloud.google.com/migrate/containers/docs/migrating-overview

Claim 1	Accused Instrumentalities
	 https://cloud.google.com/architecture/best-practices-for-building-containers One method of packaging an application into a container is with the use of a Dockerfile. The Dockerfile is similar to a script which instructs the daemon on how to assemble the container image. See the Dockerfile reference documentation) for more information. Using the Dockerfile method to build a container requires direct knowledge about the application in order to assemble the container. The first step to creating a Dockerfile is selecting an image that will be used as the basis of your image.This image should be a parent or base image maintained and published by a trusted source, usually your company. https://codelabs.developers.google.com/developing-containers-with-dockerfiles#2

Claim 1	Accused Instrumentalities
[1f] iii) wherein a SLCSE related to a predetermined function is provided to the first of the plurality of software applications for running a first instance of the SLCSE, and wherein a SLCSE for performing a same function is provided to the second of the plurality of software applications for running a second instance of the SLCSE simultaneously.	In each Accused Instrumentality, a SLCSE related to a predetermined function is provided to the first of the plurality of software applications for running a first instance of the SLCSE, and wherein a SLCSE for performing a same function is provided to the second of the plurality of software applications for running a second instance of the SLCSE simultaneously. For example, in Docker or Kubernetes containers, each container operates independently, and a base image includes essential system files, libraries, and dependencies (i.e., SLCSEs) required to run the software application within the container. Based on information and belief, each element, such as system files, libraries, and dependencies (i.e., SLCSE) is associated with an execution of a predetermined function related to the application. When an image is used to create a container in the Accused Instrumentality, an instance of the SLCSE is provided to a software application. Therefore, different instances of the SLCSE are provided to different applications for performing either a same or a different function, simultaneously. *See, e.g.*: Containers solve the portability problem by isolating the application and its dependencies so they can be moved seamlessly between machines. A process running in a container lives isolated from the underlying environment. You control what it can see and what resources it can access. This helps you use resources more efficiently and not worry about the underlying infrastructure. The container image specifies the container's file system. For example, if you're running a Node.js application, the container image would contain your app, Node.js, and other dependencies like Linux system libraries (except the kernel). A container image usually extends a base operating system image, or **base image**. This base image is the basis of your container, so you'll want to ensure that it's properly patched and free from known vulnerabilities. https://services.google.com/fh/files/misc/why_container_security_matters.pdf

Claim 1	Accused Instrumentalities
	 https://cloud.google.com/architecture/best-practices-for-building-containers

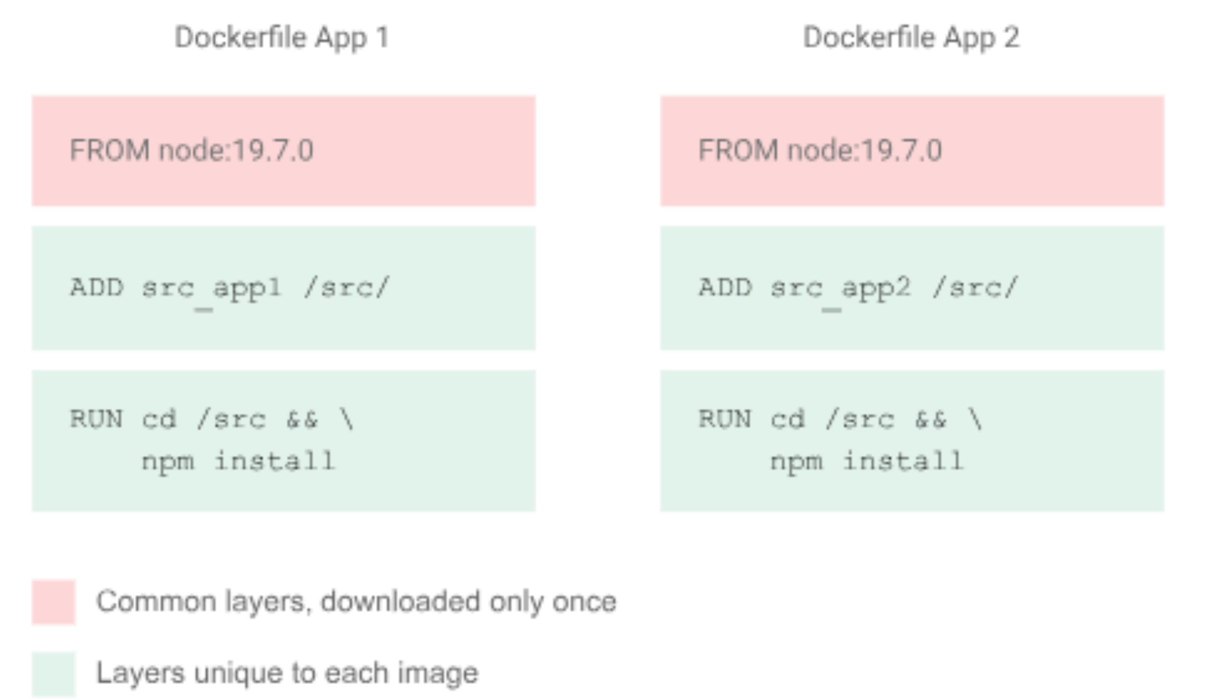
<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>

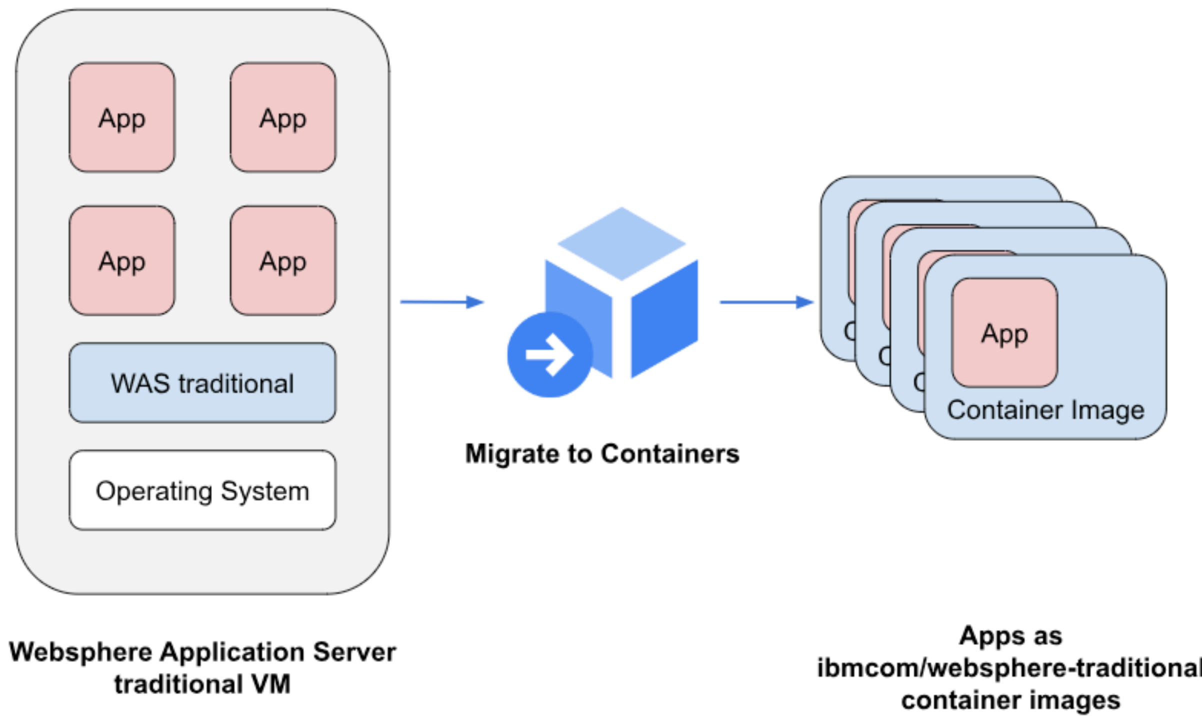
https://cloud.google.com/migrate/containers/docs/migrating-overview</td></tr>
</table>

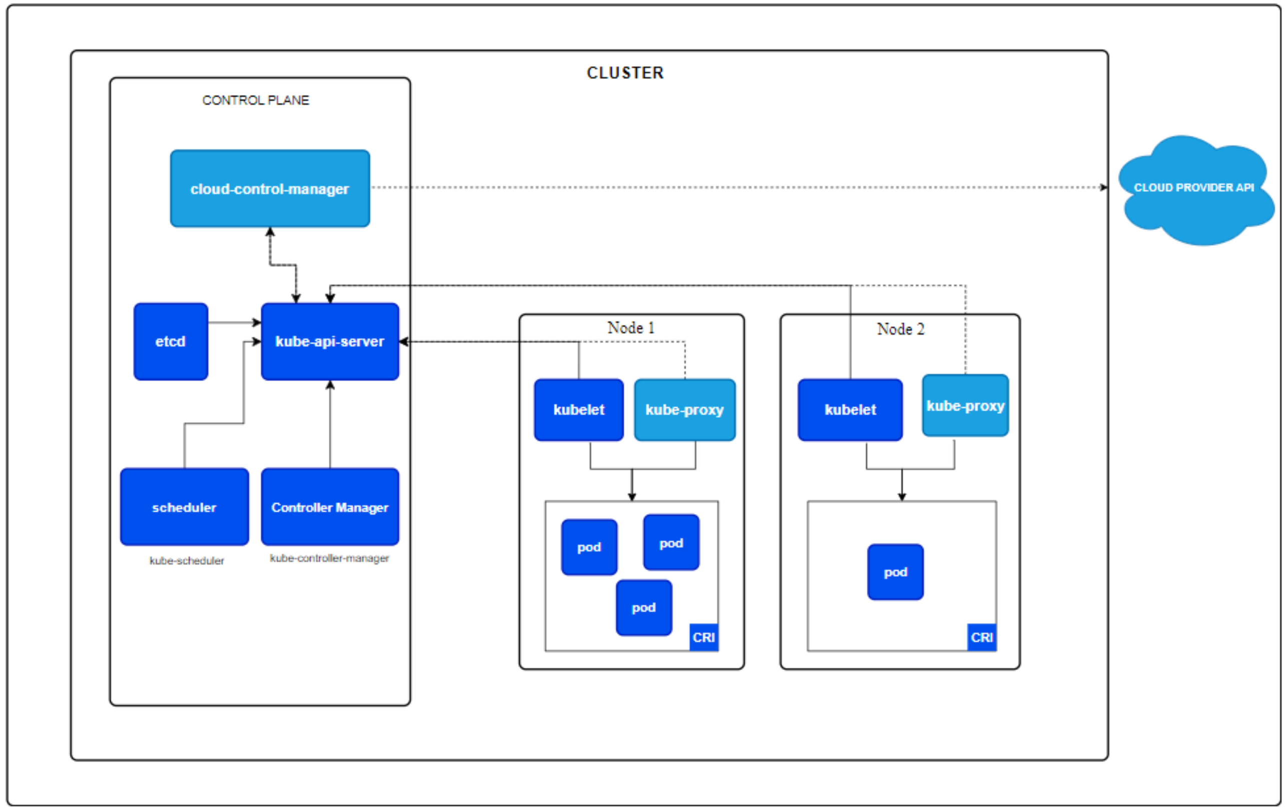
Claim 2

Claim 2	Accused Instrumentalities
2. A computing system as defined in claim 1, wherein in operation, multiple instances of an SLCSE stored in the shared library run	Each Accused Instrumentality comprises or constitutes a computing system as defined in claim 1, wherein in operation, multiple instances of an SLCSE stored in the shared library run simultaneously within the operating system.

Claim 2	Accused Instrumentalities
simultaneously within the operating system.	For example, an individual host/node runs multiple containers and/or pods simultaneously, each of which has an instance of an SLCSE. *See, e.g.*: Containers solve the portability problem by isolating the application and its dependencies so they can be moved seamlessly between machines. A process running in a container lives isolated from the underlying environment. You control what it can see and what resources it can access. This helps you use resources more efficiently and not worry about the underlying infrastructure. The container image specifies the container's file system. For example, if you're running a Node.js application, the container image would contain your app, Node.js, and other dependencies like Linux system libraries (except the kernel). A container image usually extends a base operating system image, or **base image**. This base image is the basis of your container, so you'll want to ensure that it's properly patched and free from known vulnerabilities. https://services.google.com/fh/files/misc/why_container_security_matters.pdf

Claim 2	Accused Instrumentalities
	 https://cloud.google.com/architecture/best-practices-for-building-containers

Claim 2	Accused Instrumentalities
	 https://cloud.google.com/migrate/containers/docs/migrating-overview

Claim 2	Accused Instrumentalities
	 Kubernetes cluster architecture https://kubernetes.io/docs/concepts/architecture/

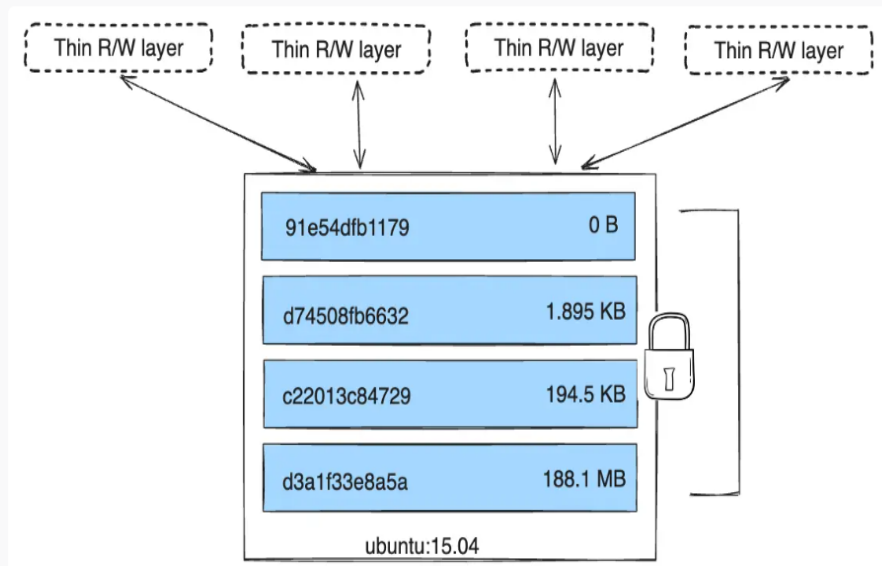
Claim 2	Accused Instrumentalities
	# Containers Each container that you run is repeatable; the standardization from having dependencies included means that you get the same behavior wherever you run it. Containers decouple applications from the underlying host infrastructure. This makes deployment easier in different cloud or OS environments. Each node in a Kubernetes cluster runs the containers that form the Pods assigned to that node. Containers in a Pod are co-located and co-scheduled to run on the same node. https://kubernetes.io/docs/concepts/containers/

Claim 2	Accused Instrumentalities
	# Kubernetes Scheduler In Kubernetes, *scheduling* refers to making sure that Pods are matched to Nodes so that Kubelet can run them. ## Scheduling overview A scheduler watches for newly created Pods that have no Node assigned. For every Pod that the scheduler discovers, the scheduler becomes responsible for finding the best Node for that Pod to run on. The scheduler reaches this placement decision taking into account the scheduling principles described below. If you want to understand why Pods are placed onto a particular Node, or if you're planning to implement a custom scheduler yourself, this page will help you learn about scheduling. https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/

Claim 2	Accused Instrumentalities
	Running containers Docker runs processes in isolated containers. A container is a process which runs on a host. The host may be local or remote. When you execute `docker run`, the container process that runs is isolated in that it has its own file system, its own networking, and its own isolated process tree separate from the host. https://docs.docker.com/engine/reference/run/

Claim 3

Claim 3	Accused Instrumentalities
3. A computing system according to claim 1 wherein OSCSEs corresponding to and capable of performing the same function as SLCSEs remain in the operating system kernel.	Each Accused Instrumentality comprises or constitutes a computing system according to claim 1 wherein OSCSEs corresponding to and capable of performing the same function as SLCSEs remain in the operating system kernel. For example, both Docker and Kubernetes systems preserve the host kernel substantially unchanged; therefore the OSCSEs corresponding to the SLCSEs remain in the operating system kernel. *See, e.g.*: Most base images are basic or minimal Linux distributions: Debian, Ubuntu, Redhat, Centos, or Alpine. Developers usually consume these images directly from Docker Hub, or other sources. There are official providers along with a wide variety of other downstream repackagers that layer software to meet customer needs. https://cloud.google.com/software-supply-chain-security/docs/base-images

Claim 3	Accused Instrumentalities
	Container image files are complete, static and executable versions of an application or service and differ from one technology to another. Docker images are made up of multiple layers, which start with a base image that includes all of the dependencies needed to execute code in a container. Each image has a readable/writable layer on top of static unchanging layers. Because each container has its own specific container layer that customizes that specific container, underlying image layers can be saved and reused in multiple containers. An Open Container Initiative (OCI) https://www.techtarget.com/searchitoperations/definition/container-containerization-or-container-based-virtualization Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state. The diagram below shows multiple containers sharing the same Ubuntu 15.04 image.  https://docs.docker.com/storage/storagedriver/

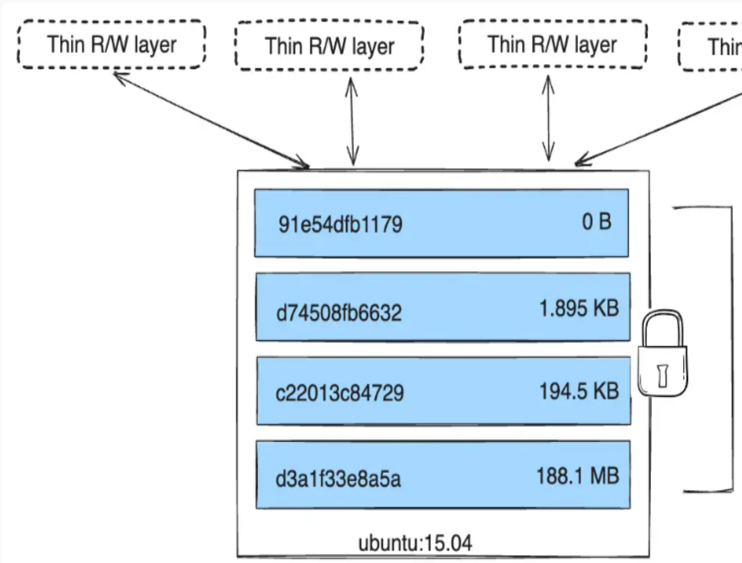
Claim 4

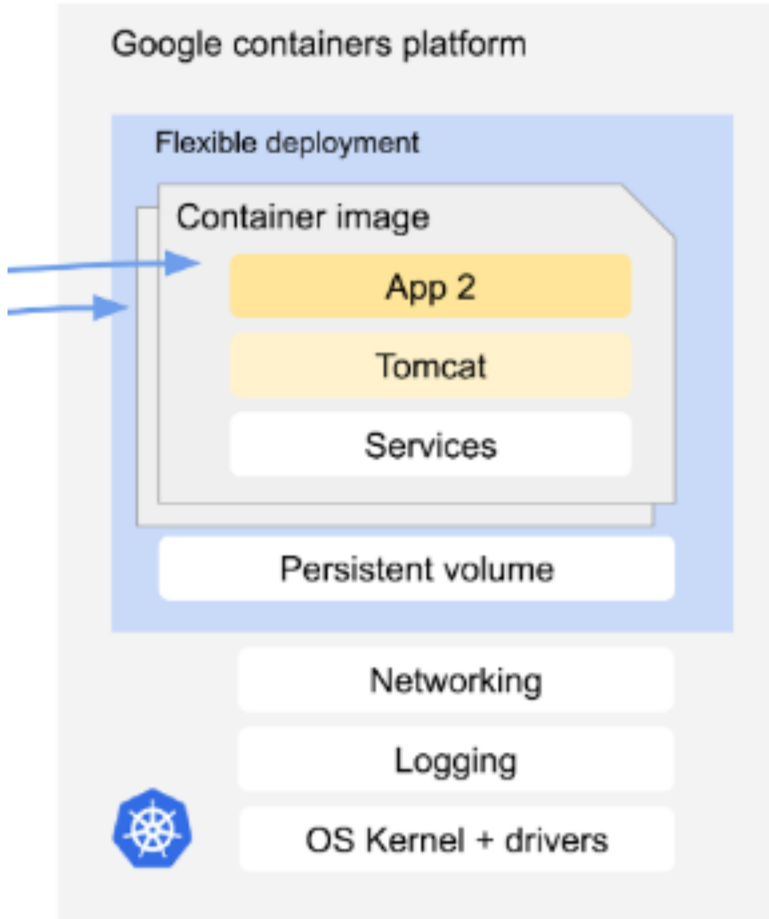
Claim 4	Accused Instrumentalities
4. A computing system according to claim 1 wherein the one or more SLCSEs provided to one of the plurality of software applications having exclusive use thereof, use system calls to access services in the operating system kernel.	Each Accused Instrumentality comprises or constitutes a computing system according to claim 1 wherein the one or more SLCSEs provided to one of the plurality of software applications having exclusive use thereof, use system calls to access services in the operating system kernel. For example, the SLCSEs in a container use system calls to access services in the operating system kernel. For example, the glibc library (or other similar library) in the container uses system calls to interface with the host Linux kernel. In general, system calls can be observed using a tool such as strace. *See, e.g.*: The **GNU C Library**, commonly known as **glibc**, is the GNU Project implementation of the C standard library. It is a wrapper around the system calls of the Linux kernel for application use. Despite its name, it now also directly supports C++ (and, indirectly, other programming languages). It was started in the 1980s by the Free Software Foundation (FSF) for the GNU operating system. https://en.wikipedia.org/wiki/Glibc

	We can now get the process id directly from the cgroup. It will be located in the `cgroup.procs` file.
	
	`### Terminal 2 - Worker Node ###`
	
	`# Get the process id`
	`$ cat cri-containerd-ceeeef06afe89c8223d33b11e8d9e0b207118ac4dac3af826687668ee1ee`
	`16254`
	
	`# Validate what is running under the process`
	`$ ps aux | grep 16254`
	`azureus+    16254 0.0  0.1 713972 10476 ?        Ssl  15:04   0:00 ./faultyapp`
	`azureus+   94806  0.0  0.0   7004  2168 pts/0    S+   16:22   0:00 grep --color=a`
	
	Got it! With that, we can try to find out what is going out inside the app. Lets try to run `strace` to get some more insight.
	
	`### Terminal 2 - Worker Node ###`
	
	`$ sudo strace -p 16254 -f`
	`...`
	`# The app is trying to read a file port.txt`
	`[pid 16269] openat(AT_FDCWD, "port.txt", O_RDONLY|O_CLOEXEC <unfinished ...>`
	`[pid 16254] epoll_pwait(5,  <unfinished ...>`
	`# The file does not exist`
	`[pid 16269] <... openat resumed>)       = -1 ENOENT (No such file or directory)`
	`[pid 16254] <... epoll_pwait resumed>[], 128, 0, NULL, 0) = 0`
	`[pid 16269] write(1, "Something went wrong...\\n", 24 <unfinished ...>`
	
	After filtering the output, we can see the application is trying to read a text file called `port.txt`, and a few lines later, there is a message stating `ENOENT (No such file or directory)`. Let's create that file.
	https://www.berops.com/blog/a-different-method-to-debug-kubernetes-pods

Claim 5

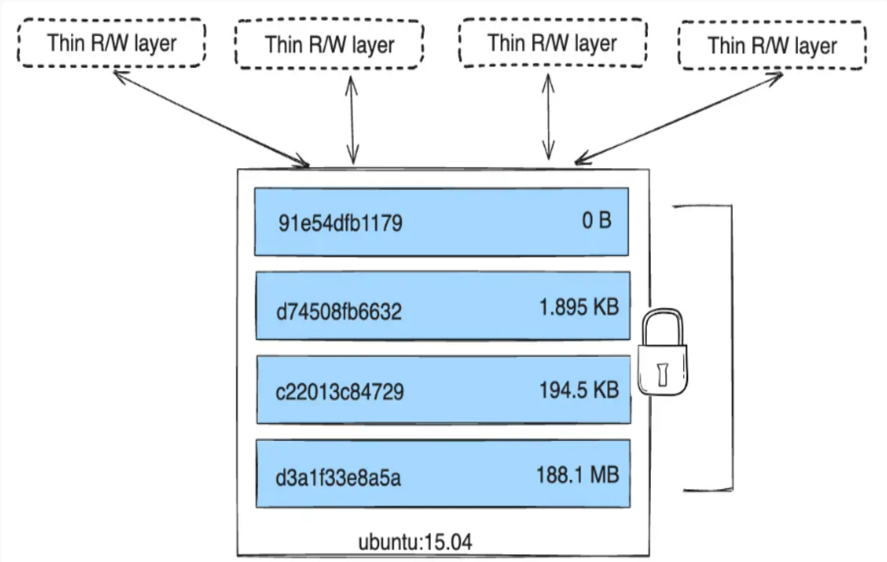
Claim 5	Accused Instrumentalities
5. A computing system according to claim 1 wherein the operating system kernel comprises a kernel module adapted to serve as an interface between an SLCSE in the context of an application program and a device driver.	Each Accused Instrumentality comprises or constitutes a computing system according to claim 1 wherein the operating system kernel comprises a kernel module adapted to serve as an interface between an SLCSE in the context of an application program and a device driver. For example, the server (node) includes an operating system having a kernel. The kernel comprises a kernel module which enables applications (including their libraries) to have access to system resources such as storage, *i.e.*, acts as an interface between applications/libraries and OS libraries or drivers *See, e.g.*: Container images A container image is a ready-to-run software package containing everything needed to run an application: the code and any runtime it requires, application and system libraries, and default values for any essential settings. https://kubernetes.io/docs/concepts/containers/ Docker is used to create, run and deploy applications in containers. A Docker image contains application code, libraries, tools, dependencies and other files needed to make an application run. When a user runs an image, it can become one or many instances of a container. ttps://www.techtarget.com/searchitoperations/definition/Docker-image

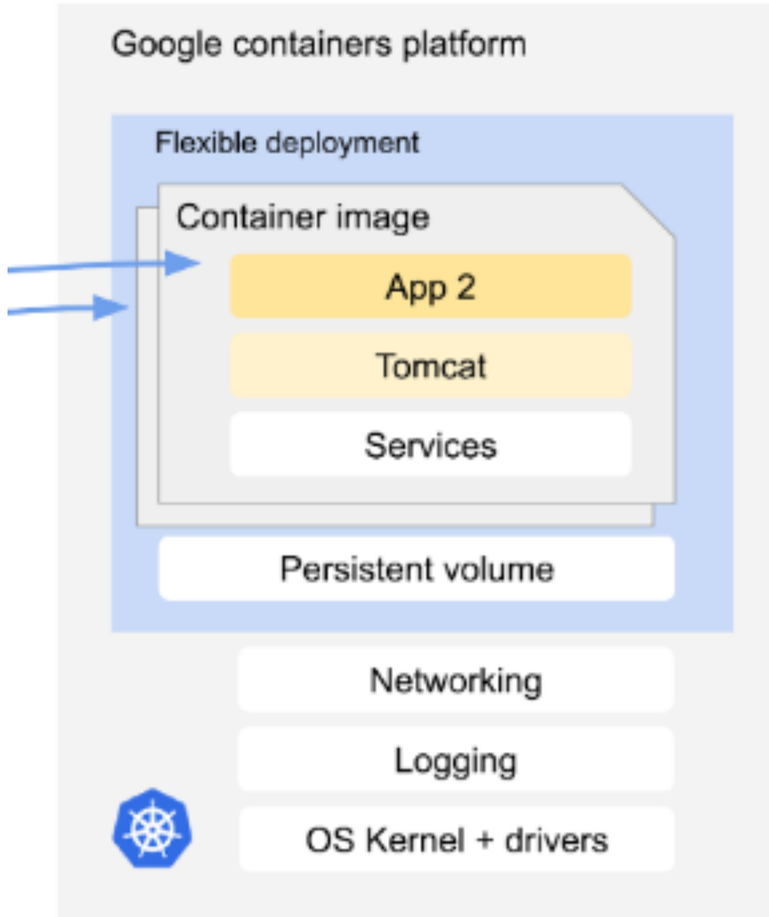
Claim 5	Accused Instrumentalities
	Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state. The diagram below shows multiple containers sharing the same Ubuntu 15.04 image.  https://docs.docker.com/storage/storagedriver/ Containers use specific features of the Linux kernel that "trick" individual applications into thinking they're in their own unique environment, even though multiple applications share the same host kernel. (If you're not familiar with the Linux kernel, it's a part of the operating system that communicates between processes--requests that do user tasks like opening a file, running a program-- and the hardware. It manages resources like memory and CPU to meet these requests). https://services.google.com/fh/files/misc/why_container_security_matters.pdf

Claim 5	Accused Instrumentalities
	 https://cloud.google.com/blog/products/application-modernization/shift-your-apps-to-container-based-workloads-on-the-command-line

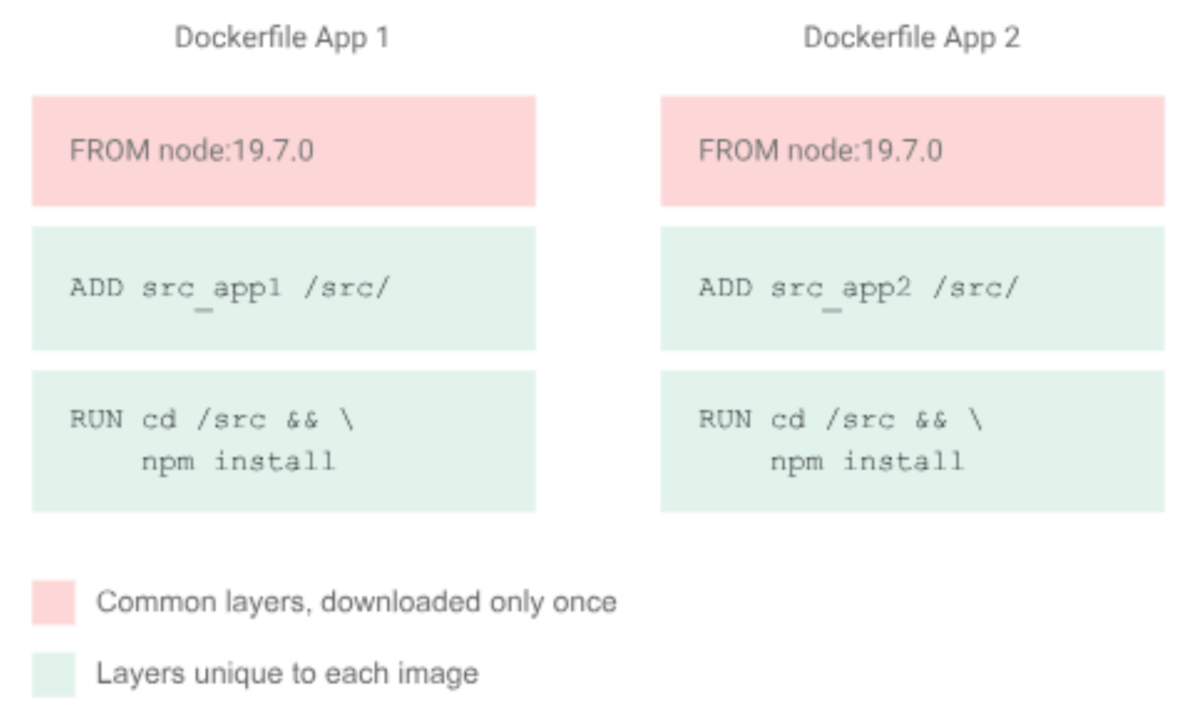
Claim 10

Claim 10	Accused Instrumentalities
10. A computing system according to claim 2 wherein SLCSEs stored in the shared library are linked to particular software applications of the plurality of software applications as the particular software applications are loaded such that the particular software applications have a link that provides unique access to a unique instance of a CSE.	Each Accused Instrumentality comprises or constitutes a computing system according to claim 2 wherein SLCSEs stored in the shared library are linked to particular software applications of the plurality of software applications as the particular software applications are loaded such that the particular software applications have a link that provides unique access to a unique instance of a CSE. For example, the containers can share common dependencies and components using layered images, and multiple containers can use the same base image. Therefore, each container, containing the application software running under the operating system of the server hosting GKE, uses a unique instance of the corresponding critical system element to execute the respective application software and has a link to that unique instance. *See, e.g.*: Container images A container image is a ready-to-run software package containing everything needed to run an application: the code and any runtime it requires, application and system libraries, and default values for any essential settings. https://kubernetes.io/docs/concepts/containers/ Docker is used to create, run and deploy applications in containers. A Docker image contains application code, libraries, tools, dependencies and other files needed to make an application run. When a user runs an image, it can become one or many instances of a container. ttps://www.techtarget.com/searchitoperations/definition/Docker-image

Claim 10	Accused Instrumentalities
	Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state. The diagram below shows multiple containers sharing the same Ubuntu 15.04 image.  https://docs.docker.com/storage/storagedriver/ Containers use specific features of the Linux kernel that "trick" individual applications into thinking they're in their own unique environment, even though multiple applications share the same host kernel. (If you're not familiar with the Linux kernel, it's a part of the operating system that communicates between processes--requests that do user tasks like opening a file, running a program-- and the hardware. It manages resources like memory and CPU to meet these requests). https://services.google.com/fh/files/misc/why_container_security_matters.pdf

Claim 10	Accused Instrumentalities
	 https://cloud.google.com/blog/products/application-modernization/shift-your-apps-to-container-based-workloads-on-the-command-line

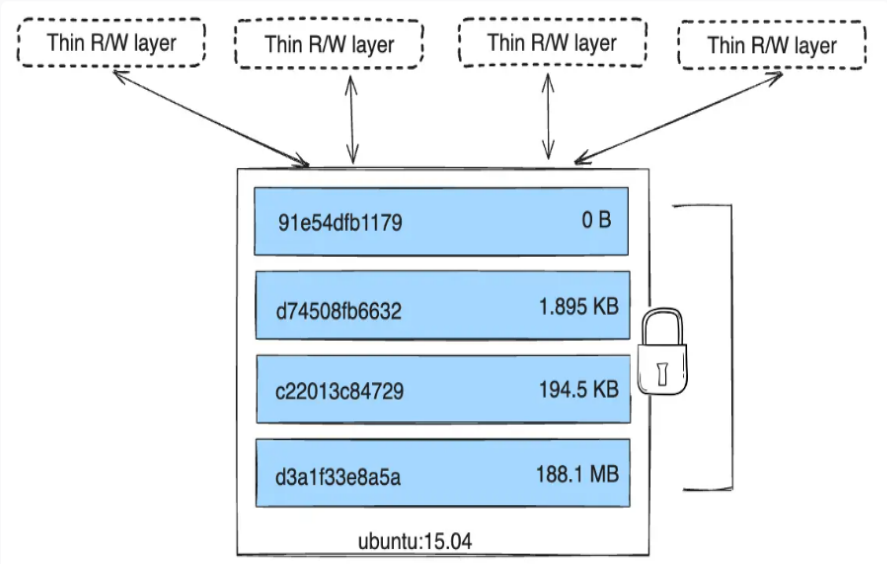
Claim 10	Accused Instrumentalities
	Containers solve the portability problem by isolating the application and its dependencies so they can be moved seamlessly between machines. A process running in a container lives isolated from the underlying environment. You control what it can see and what resources it can access. This helps you use resources more efficiently and not worry about the underlying infrastructure. The container image specifies the container's file system. For example, if you're running a Node.js application, the container image would contain your app, Node.js, and other dependencies like Linux system libraries (except the kernel). A container image usually extends a base operating system image, or **base image**. This base image is the basis of your container, so you'll want to ensure that it's properly patched and free from known vulnerabilities. https://services.google.com/fh/files/misc/why_container_security_matters.pdf

Claim 10	Accused Instrumentalities
	 https://cloud.google.com/architecture/best-practices-for-building-containers

Claim 18

Claim 18	Accused Instrumentalities
18. A computer system as defined in claim 2 wherein SLCSEs are not copies of OSCSEs.	Each Accused Instrumentality comprises or constitutes a computer system as defined in claim 2 wherein SLCSEs are not copies of OSCSEs. For example, in a typical case the SLCSEs come from a Linux distribution independent of the host operating system, and thus are not identical to the OSCSEs. For another example, the SLCSEs are

Claim 18	Accused Instrumentalities
	provided to the computer system through a separate process than the process by which the OSCSEs are provided to the computer system, and thus are not copied from the OSCSEs. *See, e.g.*: Container images A container image is a ready-to-run software package containing everything needed to run an application: the code and any runtime it requires, application and system libraries, and default values for any essential settings. https://kubernetes.io/docs/concepts/containers/ Docker is used to create, run and deploy applications in containers. A Docker image contains application code, libraries, tools, dependencies and other files needed to make an application run. When a user runs an image, it can become one or many instances of a container. ttps://www.techtarget.com/searchitoperations/definition/Docker-image

Claim 18	Accused Instrumentalities
	Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state. The diagram below shows multiple containers sharing the same Ubuntu 15.04 image.  https://docs.docker.com/storage/storagedriver/